C

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

US20020035628A1

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
14 March 2002 (14.03.2002)

PCT

(10) International Publication Number
WO 02/21296 A1

(51) International Patent Classification⁷: G06F 15/16

(21) International Application Number: PCT/US01/27394

(22) International Filing Date: 4 September 2001 (04.09.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/230,759	7 September 2000 (07.09.2000)	US
09/931,223	16 August 2001 (16.08.2001)	US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier applications:

US	60/230,759 (CON)
Filed on	7 September 2000 (07.09.2000)
US	09/931,223 (CON)
Filed on	16 August 2001 (16.08.2001)

(71) Applicant (*for all designated States except US*): MAZU NETWORKS, INC. [US/US]; 6th Floor, 125 Cambridge Park Drive, Cambridge, MA 02140 (US).

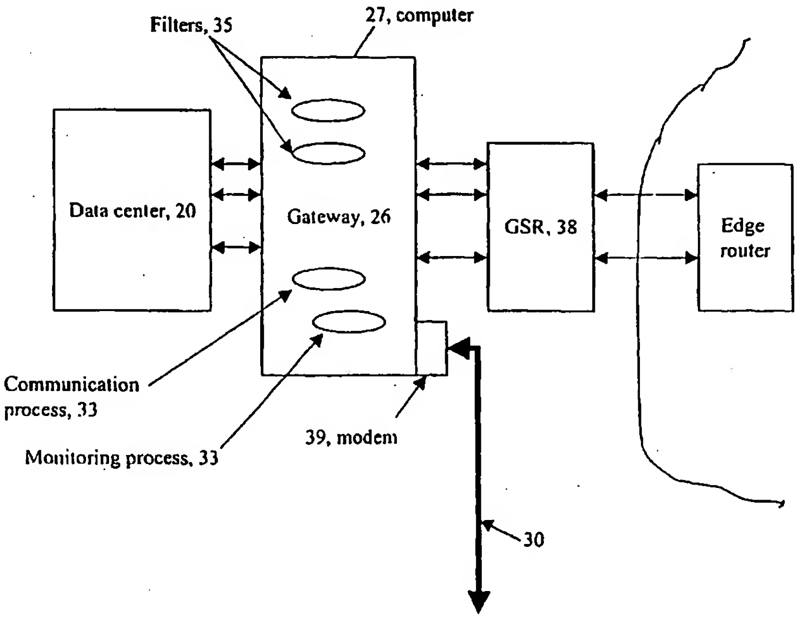
(72) Inventors; and
(75) Inventors/Applicants (*for US only*): GIL, Thomer, Michael [NL/NL]; Sarphatistraat 590a, NL-1018 AV Amsterdam (NL). POLETTO, Massimiliano, Antonio [IT/US]; 474 Broadway 6, Cambridge, MA 02138 (US). KOHLER, Edward, W., Jr. [US/US]; 805 57th Street, Oakland, CA 94608 (US).

(74) Agent: MALONEY, Denis, G.; Fish & Richardson, P.C., 225 Franklin Street, Boston, MA 02110-2804 (US).

(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

[Continued on next page]

(54) Title: STATISTICS COLLECTION FOR NETWORK TRAFFIC



(57) Abstract: A system architecture for thwarting denial of service attacks on a victim data center (20). The system includes a first plurality of monitors (33) that monitor network traffic flow through the network (30). The first plurality of monitors (33) is disposed at a second plurality of points (28) in the network (30). The system includes a central controller (27) that receives data from the plurality of monitors (33) over a network (30). The central controller (27) analyzes network traffic statistics (35) to identify malicious network traffic.

16

WO 02/21296 A1

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

STATISTICS COLLECTION FOR NETWORK TRAFFIC

Background

This invention relates to techniques to thwart network-related denial of service attacks.

In denial of service attacks, an attacker sends a large volume of malicious traffic to a victim. In one approach an attacker, via a computer system connected to the Internet infiltrates one or a plurality of computers at various data centers. Often the attacker will access the Internet through an Internet Service Provider (ISP). The attacker by use of a malicious software program places the plurality of computers at the data centers under its control. When the attacker issues a command to the computers at the data centers, the machines send data out of the data centers at arbitrary times. These computers can simultaneously send large volumes of data over various times to the victim preventing the victim from responding to legitimate traffic.

Summary

According to an aspect of the present invention, a method of monitoring traffic flow in a monitor device disposed to receive network traffic packets includes producing statistics corresponding to a parameter of traffic flow to trace the source of an attack. The method also includes mapping the traffic flow into a plurality of buckets by applying a hash function "f(h)" to the parameter of the traffic flow to output an integer corresponding to one of the buckets and accumulating statistics from the packets. The method includes comparing the number of buckets to a threshold and determining whether the number of buckets should be divided into more buckets or combined into fewer buckets based on comparing the number of buckets to the threshold.

According to an additional aspect of the present invention, a computer program product residing on a computer readable for monitoring network traffic flow in a network includes instructions to cause a computer to map traffic flow into a plurality of buckets by applying a hash function "f(h)" to a parameter of the traffic flow to output an integer corresponding to one of the buckets. The program accumulates statistics from the packets and compares the accumulated statistic values from the buckets to configured threshold values corresponding to the number of buckets to determine that an event is of significance. The program adjusts the number of buckets as the number of buckets approaches a second threshold.

According to an additional aspect of the present invention, a data collector to collect statistical information about network flows includes a computer readable medium and a computing device that executes a computer program product stored on the computer readable medium. The computer program product includes instructions to cause the computing device to map traffic flow into a plurality of buckets by applying a hash function "f(h)" to the parameter of the traffic flow to output an integer corresponding to one of the buckets and accumulate statistics from the packets. The program compares the accumulated statistic values from the buckets to configured threshold values corresponding to the number of buckets to determine that an event is of significance and adjusts the number of buckets as the number of buckets approaches a second threshold. The data collector also includes a port to link the data collector to a central control center.

One or more aspects of the invention may provide some or all of the following advantages.

An attack may be designed to explode the number of buckets. Since each bucket consumes memory space, the attack can be designed to consume all available memory and crash the data collection processes and devices.

The invention prevents that type of attack setting thresholds on the number of buckets. As buckets approach a bucket threshold "Th", the process can take several buckets and divide them in more buckets or combine them into fewer buckets. The variable number of buckets dynamically adjusts the monitoring process to the amount of traffic and number of flows, so that monitoring is not vulnerable to a DoS attacks against its own resources.

The variable number of buckets also helps to efficiently identify the source(s) of attack by breaking down traffic into different categories (buckets) and looking at the appropriate parameters and thresholds in each bucket. With the invention, traffic is monitored at multiple levels of granularity, from aggregate to individual flows. The invention can be applied to various types of monitoring, e.g., TCP packet ratios, repressor traffic, and so forth.

Brief description of the drawings

FIG. 1 is a block diagram of networked computers showing an architecture to thwart denial of service attacks.

FIG. 2 is a block diagram depicting details of placement of a gateway.

FIG. 3 is a block diagram depicting details of placement of data collectors.

FIG. 4 is flow chart depicting a data collection process.

FIG. 5 is a flow chart depicting details of a control center.

FIG. 6 is a diagram depicting functional layers of a monitoring process.

FIG. 7 is a diagram depicting one technique to gather statistics for use in algorithms that determine sources of an attack.

FIG. 8 is a diagram depicting an alternative technique to gather statistics for use in algorithms that determine sources of an attack.

FIG. 9 is flow chart depicting a process to determine receipt of bad TCP traffic.

FIG. 10 is flow chart depicting a process to defend against setup time connection attacks.

Detailed Description

Referring to FIG. 1, an arrangement 10 to thwart denial of service attacks (DoS attacks) is shown. The arrangement 10 is used to thwart an attack on a victim data center 12, e.g., a web site or other network site under attack. The victim 12 is coupled to the Internet 14 or other network. For example, the victim 12 has a web server located at a data center (not shown).

An attacker via a computer system 16 that is connected to the Internet e.g., via an Internet 14 Service Provider (ISP) 18 or other approach, infiltrates one or a plurality of computers at various other sites or data centers 20a-20c. The attacker by use of a malicious software program 21 that is generally surreptitiously loaded on the computers of the data centers 20a-20c, places the plurality of computers in the data centers 20a-20c under its control. When the attacker issues a command to the data centers 20a-20c, the data centers 20a-20c send data out at arbitrary times. These data centers 20a-20c can simultaneously send large volumes of data at various times to the victim 12 to prevent the victim 12 from responding to legitimate traffic.

The arrangement 10 to protect the victim includes a control center 24 that communicates with and controls gateways 26 and data collectors 28 disposed in the network 14. The arrangement protects against DoS attacks via intelligent traffic analysis and filtering that is distributed throughout the network. The control center 24 is coupled to the gateways 26 and data collectors 28 by a hardened, redundant network 30. Gateways 26 and data collectors 28 are types of monitors that monitor and collect statistics on network traffic. In preferred embodiments, the network is inaccessible to the attacker. The gateway 26 devices are located at the edges of the Internet 14, for instance, at the entry points of data centers. The gateway devices constantly analyze traffic, looking for congestion or traffic levels that indicate the onset of a DoS attack. The data collectors 28 are located *inter alia* at major peering points and network points of presence (PoPs). The data collectors 28 sample packet traffic, accumulate, and collect statistical information about network flows.

All deployed devices e.g., gateways 26 and data collectors 28 are linked to the central control center. The control center aggregates traffic information and coordinates measures to track down and block the sources of an attack. The arrangement uses a distributed analysis emphasizing the underlying characteristics of a DoS attack, i.e., congestion and slow server response, to produce a robust and comprehensive DoS solution. Thus, this architecture 10 can stop new attacks rather than some solutions that can only stop previously seen attacks. Furthermore, the distributed architecture 10 will frequently stop an attack near its source, before it uses bandwidth on the wider Internet 14 or congests access links to the targeted victim 12.

A virus is one way to get attacks started. When surfing the web page a user may download something, which contains a virus that puts the user's computer under the control of some hacker. In the future, that machine can be one of the machines that launches the attack. The attacker only needs a sufficient amount of bandwidth to get a sufficient number of requests out to the victim 12 to be malicious.

Referring to FIG. 2, details of an exemplary deployment of a gateway is shown. Other deployments are possible and the details of such deployments would depend on characteristics of the site, network, cost and other considerations. The gateway 26 is a program executing on a device, e.g., a computer 27 that is disposed at the edge of the data center 20 behind an edge router at the edge of the Internet 14. Additional details on the gateway 26 are discussed below and in the APPENDIX A. In a preferred embodiment, a plurality of gateway devices are deployed at a corresponding plurality of locations, e.g., data centers or sites over the network, e.g., the Internet 14. There can be one gateway or a plurality of gateways at each data center, but that is not necessarily required.

The gateway 26 includes a monitoring process 32 (FIG. 6B) that monitors traffic that passes through the gateway as well as a communication process 33 that can communicate statistics collected in the gateway 26 with the data center 24. The gateway uses a separate interface over a private, redundant network, such as a modem 39 to communicate with the control center 24 over the hardened network 30. Other interface types besides a modem are possible. In addition, the gateway 26 can include processes 35 to allow an administrator to insert filters to filter out, i.e., discard packets that the device deems to be part of an attack, as determined by heuristics described below.

An attack can be designed to either overload the servers or overload some part of the network infrastructure inside the victim site 12. Thus, the victim site 12 can include routers, switches, load balancers and other devices inside the data center that can be targeted by the attack. A particularly troublesome attack causes overload of upstream bandwidth. Upstream bandwidth is the capacity between the victim 12 data center 12a and one or a plurality of routers or switches belonging to the victim 12 data center's network service provider, which provides connectivity to the rest of the network, e.g., the Internet.

For an exemplary configuration, the victim site 12 can include a plurality of high bandwidth lines feeding a GSR (Gigabit Switch Router). At the output of the GSR are exit ports to various parts of the data center. The GSR is generally very high bandwidth and generally does not crash. The gateway 26 is placed behind the GSR and across some or all of the output ports of the GSR into the data center. This configuration allows the gateway 26 to monitor and control some or all of the traffic entering the data center without the need to provide routing functionality.

Alternatively, a gateway 26 can tap a network line without being deployed physically in line, and it can control network traffic, for example, by dynamically installing filters on nearby routers. The gateway 26 would install these filters on the appropriate routers via an out of band connection, i.e. a serial line or a dedicated network connection. Other arrangements are of course possible.

Referring to FIG. 3, data collectors 28 are shown coupled to the network to tap or sample traffic from data centers 20a-20c. Although data collectors 28 can be dispersed throughout the network 14 they can be strategically disposed at peering points, i.e., points where network traffic from two or more different backbone providers meet. The data collectors 28 can also be disposed at points of presence (PoPs). The data collectors 28 monitor and collect information pertaining to network traffic flow. The data collectors process statistics based on monitored network traffic that enters a peering point. Data collectors 28 include a monitoring process 32 (FIG. 6) as well as a communication process that communicates data to the control center over the hardened network 30. One or more data collector devices 28 use the monitoring process to monitor one or more lines that enter the peering point. Each data collector 28 would be able to monitor one or more lines depending on the specifics of how the network is configured and bandwidth requirements.

The gateway 26 and data collector 26 are typically software programs that are executed on devices such as computers, routers, or switches. In one arrangement, packets pass through the gateway 26 disposed at the data center 22a and are sampled by the data collector.

Referring to FIG. 4, the data collector 26 performs 40 a sampling and statistic collection process 40. The data collector samples 42 one (1) packet in every (n) packets and has counters to collect statistics about every packet. The data collector 26 parses the information in the sampled packet. Information collected includes source information 44, which may be fake or spoofed, e.g., not correct information. It will also include destination information 46, which generally is accurate information. The data collector 28 collects that information but need not log the sampled packets. The data collector 28 maintains a log over a period of time, e.g., in the last hour. As an example, the log that the data collector 26 maintains is a log that specifies that the data collector has seen a certain number of packets, e.g., 10,000 packets of a particular kind, that apparently originated from a particular source(s) that are going to a particular destination.

Based on rules 48 within the data collector 26, the data collector 26 analyzes 50 the collected statistics and may if necessary compose 52 a message that raises an alarm. Alternatively, the data collector can respond to queries concerning characteristics of traffic on the network. Typically, the queries can be for information pertaining to statistics. It can be in the form of an answer to a question e.g., how many packets of a type did the data collector see or it can be a request to down load via the hardened network, the entire contents of the log. One rule is that when the data collector 26 starts sampling, the data collector periodically logs data and produces a log of a large plurality of different network flows over a period of time.

Referring to FIG. 5, a deployment for the control center 24 is shown. The control center 24 receives information from one or more gateways 26 and data collectors 28 and performs appropriate analysis using an analysis process 62. The control center is a hardened site.

The control center 24 has multiple upstream connections so that even during an attack it will have other ways to couple to the network 30. Several approaches can be used to harden the site. One approach can use special software between the site and the Internet 14 to make it immune to attack. An approach is to have a physically separate network 30 connected to all of the devices, e.g., gateways 26 and data collectors 28. One exemplary embodiment of that physically separate network 30, which is hardened, is the telephone system. Thus, each one of the data collectors 26 and gateways 26 includes an interface to the separate network, e.g., a modem. The data center 26 also includes a corresponding interface to the separate network, e.g., a modem or a modem bank 60.

With this approach, the redundant network 30 is not accessible to the attacker. The redundant network 30 thus is available to communicate between the data center 24 and data collectors and gateways to coordinate response to an attack. In essence, the network 30 used by the data center to communicate with the data collectors 26 and gateways 26 is not available to the attacker. Alternatively, if less than complete assurance is required, the control center could be resistant to attack and still be connected to the Internet 14.

The analysis process 62 that is executed on the control center 24 analyzes data from the gateways 26 and data collectors 28. The analysis process 62 tries to detect attacks on victim sites. The analysis process 62 views attacks as belonging to, e.g., one of three classes of attack. Herein these classes of attack are denoted as low-grade with spoofing, low-grade without spoofing and high-grade whether spoofing or non-spoofing.

A low-grade attack is an attack that does not take out upstream bandwidth. A low-grade attack does not significantly overburden the links between the Internet 14 and the victim data center 12. The low-grade non-spoofing attack is the simplest type of attack to defend against. It simply requires identifying the source of the attack and a mechanism to notify an administrator at the victim site to install a filter or filters at appropriate points to discard traffic containing the source address associated with the attack.

With a low-grade spoofing-type attack, an attacker sends an IP-packet to a destination but fakes the source address. There is no way to enforce use of an accurate source address by a sender. During a spoofing attack, each one of the attacking machines will send a packet with a fake, e.g., randomly selected or generated source address. Under this type of attack, the victim 12 alone cannot thwart the attack. An administrator at the victim 12 can try to put a filter on a router to stop the packets. However, there is no way for the administrator to guess what the random address of the next packet will be.

The control center 24 also includes a communication process 63 to send data to/from the gateways 26 and data collectors 28. The gateway 26 at the victim 12 contacts the control center and notifies the control center 24 that the victim 12 data center is under a spoofing attack. The gateway 26 identifies itself by network address (e.g., static IP address if on the Internet 14), via a message to the control center 24. The message sent over the hardened network 30 indicates the type of attack, e.g., an attack from addresses that the victim 12 cannot stop because it is a spoofing type of attack. The control center queries data collectors 28 and asks which data collectors 28 are seeing suspicious traffic being sent to the victim 12.

The packets from the attacker will have faked source addresses that will be changing with time. However, the control center can issue a query for this kind of packet by victim destination address. The data collectors 28 reply with the information collected. Based on that collected information from the data collectors 28, the control center can then determine what data centers are performing the spoofing on the victim 12.

In the present configuration, there are two possible sources of attack traffic: either the attacker is behind a gateway 26 or not. If the attacker is behind a gateway 26, the control center issues a request to the appropriate gateway 26 to block the attacking traffic, e.g. by allowing the appropriate gateway 26 to discard traffic, e.g., packets that contain the victim 12 destination address. The gateway 26 stops that traffic in a transparent manner. If the attacker is not behind a gateway 26, data collectors 28 are used to provide information about possible locations of the attackers. The availability of information from data collectors 28 increases the speed with which attackers are discovered. The data collectors 28 are positioned at network switching points that see a high volume of traffic, which minimizes the required number of deployed data collectors.

The high-grade attacks are attacks that take out the link between the victim 12 data center and the Internet 14. With a high-grade attack it does not matter whether the victim 12 is spoofed or not. Under a high-grade attack, the attack requires cooperation just like the low grade spoofing attack. Thus, the same thwarting mechanism is used for either spoofing or non-spoofing, e.g., using information from the data collectors 28 to identify attacking networks. This information is used to either automatically shutdown traffic having the victim's destination address at the appropriate gateways 26 or is used to identify networks or data centers from which the attack is originating and to follow up with calls to the appropriate administrators.

Referring to FIG. 6, a monitoring process 32 is shown. The monitoring process 32 can be deployed on data collectors 28 as well as gateways 26. The monitoring process 32 includes a process 32a to collect statistics of packets that pass by the data collectors 28 or through the gateways 26. The monitoring process 32 also includes several processes 32b to identify, malicious traffic flows based on the collected statistics as further described below.

Referring to FIG. 7, the gateways 26 and data collectors 28 are capable of looking at multiple levels of granularity. The gateways 26 and data collectors have monitoring process 32 used to measure some parameter of traffic flow. One goal of the gateways 26 and data collectors 28 is to measure some parameter of network traffic. This information collected by the gateways 26 and data collectors is used to trace the source of an attack.

One of the algorithms to measure parameters of traffic flow divides the traffic flow into buckets. For example, consider one simple parameter, the count of how many packets a data collector or gateway examines. An algorithm to track the count of this parameter starts with a predefined number of buckets, e.g., "N" buckets. The buckets are implemented as storage areas in the memory space of the data collector or gateway device. The algorithm will use some hash function "f(h)", which takes the packet and outputs an integer that corresponds to one of the buckets "B_1 - B_N". Statistics from the packets start accumulating in the buckets "B_1 - B_N". The buckets "B_1 - B_N" are configured with threshold values "Th." As the contents of the buckets B_1 - B_N reach the configured thresholds values "Th", (e.g., compare values of packet count or packet rate to threshold), the monitoring process 32 deems that event to be of significance. The monitoring process 32 takes that bucket, e.g., B_i and divides that bucket B_i into some other number M of new buckets B_{i1} - B_{iM}. Each of the new buckets B_{i1} - B_{iM} contains values appropriately derived from the original bucket B_i. Also, the hash function is extended to map to N+M-1 "h→N+M-1" values, rather than the original N values.

An attack designed to use the algorithm of FIG. 6 against a gateway 26 or a data collector 28 might send packets in such a fashion as to explode the number of buckets. Since each bucket consumes memory space, the attack can be designed to consume all available memory and crash the device, e.g., computer on which the monitoring process 32 executes. There are ways of preventing that type of attack on the monitoring process 32. One way is to make the hash function change periodically, e.g., randomly. Also the hash function is secret so that the packets are reassigned to different buckets in ways unknown to the attackers.

Referring to FIG. 8, a second method is that instead of using just thresholds and values inside a given bucket, the monitoring process 32 also sets thresholds on the number of buckets. As the gateway 26 or data collector 28 approaches a bucket threshold "Th", the gateway 26 or data collector 28 have the ability to take several buckets B_1 - B_3 and divide them in more buckets B_1 - B_4 or combine them into fewer bucket B_1 - B_2.

The function of the variable number of buckets is to dynamically adjust the monitoring process to the amount of traffic and number of flows, so that the monitoring device (e.g., gateway 26 or data collector 28) is not vulnerable to DoS attacks against its own resources. The variable number of buckets also efficiently identifies the source(s) of attack by breaking down traffic into different categories (buckets) and looking at the appropriate parameters and thresholds in each bucket.

Thus, with multi-level analysis as discussed in FIGS. 6 and 7, traffic is monitored at multiple levels of granularity, from aggregate to individual flows. Multi-level analysis can be applied to all types of monitoring (i.e. TCP packet ratios, repressor traffic, etc. discussed below) except TCP SYN proxying (because the latter requires per-connection monitoring of all half-open connections as discussed below).

The monitoring process 32 has the gateway 26 or the data collectors 28 keep track of a metric (such as packet ratio) for each of n traffic buckets. (If n=1, the monitoring process 32 tracks the metric for all traffic in the aggregate.) The monitoring process 32 places packets into buckets according to a hash function of the source or destination address. If the metric in any bucket exceeds a given "suspicious" threshold, that bucket is split into several smaller buckets, and the metric is tracked individually for each new bucket. In the limit, each bucket can correspond to a single flow (source address/port and destination address/port pair). The resulting per-flow monitoring is resilient to denial-of-service attacks. If the number of buckets exceeds a given memory limit (for example, due to a many-flow spoofing attack), several fine-grain buckets can be aggregated into a single coarse-grain bucket. The hash function for placing packets into traffic buckets is secret and changes periodically, thwarting attacks based on carefully chosen addresses.

In the worst case, an attacker actually spoofs packets from all possible addresses. An IP address, for example is 32 bits long. This address length allows for approximately 4 billion possible random addresses and makes it impossible for the gateway at the victim site 12 to identify the attacker. In that worst case, the gateway 26 calls the control center, indicates the address of the gateway 26, and conveys that the gateway 26 is receiving unreasonably high levels of random traffic. The control center 24 contacts the data collectors 28. The control center 24 analyzes the statistics collected by the data collectors 28 to try to determine the source of the traffic.

Egress filtering is a recommended Internet 14 best practice procedure that does not allow any packets out of a network unless the source address belongs to that network. Egress filtering prevents hosts on that network from sending out packets with completely random source addresses. Rather, the space of usable fake addresses is limited by the size of the host's network address space, and may range up to 24 bits rather than the full 32 bits. If an attacker is attacking from a network that performs egress filtering, then all the attack traffic reaching a victim will fall into a smaller number of buckets, those corresponding to the source network address. In this way, the gateway 26 can identify the approximate source of the attack without necessarily relying on the control center or data collectors.

Several methods can be used separately or in combination to identify, malicious traffic flows. For example, the gateway 26 can detect DoS attacks and identify malicious flows or source addresses using at least one or more of the following methods including: analyzing packet ratios of TCP-like traffic; analyzing "repressor" traffic for particular types of normal traffic; performing TCP handshake analysis; performing various types of packet analysis at packet layers 3-7; and logging/historical analysis.

Packet ratios for TCP-like traffic.

The Transmission Control Protocol (TCP) is a protocol in which a connection between two hosts, a client C, e.g. a web browser, and a server S, e.g. a web server, involves packets traveling in both directions, between C and S and between S and C. When C sends data to S and S receives it, S replies with an ACK ("acknowledgement") packet. If

C does not receive the ACK, it will eventually try to retransmit the data to S, to implement TCP's reliable delivery property. In general, a server S will acknowledge (send an ACK) for every packet or every second packet.

Referring to FIG. 9, the monitoring process in the gateway 26 can examine 82 a ratio of incoming to outgoing TCP packets for a particular set of machines, e.g. web servers. The monitoring process can compare 84 the ratio to a threshold value. The monitoring process can store 86 this ratio, time stamp it, etc. and conduct an ongoing analysis 88 to determine over time for example how much and how often it exceeds that ratio. As the ratio grows increasingly beyond 2:1, e.g., to about 3:1 or so, it is an increasing indication that the machines are receiving bad TCP traffic, e.g. packets that are not part of any established TCP connection, or that they are too overloaded to acknowledge the requests. This ratio is one of the parameters measured using the multiple-bucket algorithm described previously.

The gateway 26 divides traffic into multiple buckets, e.g. by source network address, and tracks the ratio of ingoing to outgoing traffic for each bucket. As the ratio for one bucket becomes skewed, the gateway 26 may subdivide that bucket to obtain a more detailed view. The gateway 26 raises 90 a warning or alarm to the data center 24 and/or to the administrators at the victim site 12.

Repressor traffic

The phrase "repressor traffic" as used herein refers to any network traffic that is indicative of problems or a potential attack in a main flow of traffic. A gateway 26 may use repressor traffic analysis to identify such problems and stop or repress a corresponding attack.

One example of repressor traffic is ICMP port unreachable messages. These messages are generated by an end host when it receives a packet on a port that is not responding to requests. The message contains header information from the packet in question. The gateway 26 can analyze the port unreachable messages and use them to generate logs for forensic purposes or to selectively block future messages similar to the ones that caused the ICMP messages.

TCP handshake analysis

A TCP connection between two hosts on the network is initiated via a three-way handshake. The client, e.g. C, sends the server, e.g. S, a SYN ("synchronize") packet. S the server replies with a SYN ACK ("synchronize acknowledgment") packet. The client C replies to the SYN ACK with an ACK ("acknowledgment") packet. At this point, appropriate states to manage the connection are established on both sides.

During a TCP SYN flood attack, a server is sent many SYN packets but the attacking site never responds to the corresponding SYN ACKs with ACK packets. The resulting "half-open" connections take up state on the server and can prevent the server from opening up legitimate connections until the half-open connection expires, which usually takes 2-3 minutes. By constantly sending more SYN packets, an attacker can effectively prevent a server from serving any legitimate connection requests.

Referring to FIG. 10, in an active configuration, a gateway 26 can defend against SYN flood attacks. During connection setup, the gateway forwards 102 a SYN packet from a client to a server. The gateway forwards 104 a resulting SYN ACK packet from a server to client and immediately sends 106 ACK packet to the server, closing a three-way handshake. The gateway maintains the resulting connection for a timeout period 108. If the ACK packet does not arrive from client to server 110, the gateway sends 112 a RST ("reset") to the server to close the connection. If the ACK arrives 114, gateway forwards 116 the ACK and forgets 118 about the connection, forwarding subsequent packets for that connection. A variable timeout 120 period can be used. The variable time out period can be inversely proportional to number of connections for which a first ACK packet from client has not been received. If gateway 26 is placed inline in the network, when number of non-ACK'ed connections reaches a configurable threshold 122, the gateway will not forward any new SYNs until it finishes sending RSTs for those connections.

In a passive configuration, a gateway 26 can similarly keep track of ratios of SYNs to SYN ACKs and SYN ACKs to ACKs, and raise appropriate alarms when a SYN flood attack situation occurs.

Layer 3-7 analysis.

With layer 3-7 analysis, the gateway 26 looks at various traffic properties at network packet layers 3 through 7 to identify attacks and malicious flows. These layers are often referred to as layers of the Open System Interconnection (OSI) reference model and are network, transport, session, presentation and application layers respectively. Some examples of characteristics that the gateway may look for include:

1. Unusual amounts of IP fragmentation, or fragmented IP packets with bad or overlapping fragment offsets.

2. IP packets with obviously bad source addresses, or ICMP packets with broadcast destination addresses.

3. TCP or UDP packets to unused ports.

4. TCP segments advertizing unusually small window sizes, which may indicate load on server, or TCP ACK packets not belonging to a known connection.

5. Frequent reloads that are sustained at a rate higher than plausible for a human user over a persistent HTTP connection.

Logging and historical traffic analysis

The gateways 26 and data collectors 28 keep statistical summary information of traffic over different periods of time and at different levels of detail. For example, a gateway 26 may keep mean and standard deviation for a chosen set of parameters across a chosen set of time-periods. The parameters may include source and destination host or network addresses, protocols, types of packets, number of open connections or of packets sent in either direction, etc. Time periods for statistical aggregation may range from minutes to weeks. The device will have configurable thresholds and will raise warnings when one of the measured parameters exceeds the corresponding threshold.

The gateway 26 can also log packets. In addition to logging full packet streams, the gateway 26 has the capability to log only specific packets identified as part of an attack (e.g., fragmented UDP packets or TCP SYN packets that are part of a SYN flood attack). This feature of the gateway 26 enables administrators to quickly identify the important properties of the attack.

Building a DoS-resistant network

The network of gateways 26, data collectors 28, and control center 24 are made DoS resistant by combining and applying several techniques. These techniques include the use of SYN cookies and "hashcash" to make devices more resistant to SYN floods and other attacks that occur at connection setup time. Also, the data center can use authentication and encryption for all connections. Private/public key pairs are placed on machines before deployment to avoid man-in-the-middle attacks. The control center 24 can have multiple physical connections from different upstream network service providers. The network over which the data center communicates between gateways and data collectors is a private redundant network that is inaccessible to attackers.

Information exchange between gateways/data collectors and the control center is efficient by transferring only statistical data or minimal header information, and by compressing all data.

This application includes an APPENDIX A attached hereto and incorporated herein by reference. APPENDIX A includes Click code for monitor software.

This application also includes an APPENDIX B attached hereto and incorporated herein by reference. APPENDIX B sets out additional modules for a Click Router that pertains to thwarting DoS attacks. "Click" is a modular software router system developed by The Massachusetts Institute of Technology's Parallel and Distributed Operating Systems group. A Click router is an interconnected collection of modules or elements used to control a router's behavior when implemented on a computer system.

Other embodiments are within the scope of the appended claims.

APPENDIX A

```
network monitor/defender
//
// Has two operating modes: if MONITOR is defined, it monitors the network
// instead of defending against DDoS attacks.
//
// ICMP_RATE specifies how many ICMP packets allowed per second. Default is
// 500. UDP_NF_RATE specifies how many non-fragmented UDP (and other non-
TCP
// non-ICMP) packets allowed per second. Default is 3000. UDP_F_RATE specifies
// how many fragmented UDP (and other non-TCP non-ICMP) packets allowed per
// second. Default is 1000. All the SNIFF rates specify how many bad packets
// sniffed per second.
//
// For example, if MONITOR is not defiend, and all SNIFF rates are 0, then the
// configuration defends against DDoS attacks, but does not report bad
// packets.
//
// can read:
//  - tcp_monitor: aggregate rates of different TCP packets
//  - ntcp_monitor: aggregate rates of different non TCP packets
//  - icmp_unreach_counter: rate of ICMP unreachable pkts
//  - tcp_ratemon: incoming and outgoing TCP rates, grouped by non-local hosts
//  - ntcp_ratemon: incoming UDP rates, grouped by non-local hosts
//
// Note: handles full fast ethernet, around 134,500 64 byte packets, from
// attacker.
//
//
// TODO:
//  - fragmented packet monitor

#ifndef ICMP_RATE
#define ICMP_RATE        500
#endif

#ifndef UDP_NF_RATE
#define UDP_NF_RATE      2000
#endif

#ifndef UDP_F_RATE
#define UDP_F_RATE       1000
#endif

#ifndef SUSP_SNIFF
#define SUSP_SNIFF       100      // # of suspicious pkts sniffed per sec
```

```
#endif

#ifndef TCP_SNIFF
#define TCP_SNIFF  100      // # of TCP flood pkts sniffed per sec
#endif

#ifndef ICMP_SNIFF
#define ICMP_SNIFF          75      // # of ICMP flood pkts sniffed per sec
#endif

#ifndef UDP_NF_SNIFF
#define UDP_NF_SNIFF        75      // # of non-frag UDP flood pkts sniffed per sec
#endif

#ifndef UDP_F_SNIFF
#define UDP_F_SNIFF         75      // # of frag UDP flood pkts sniffed per sec
#endif

#include "if.click"

#include "sampler.click"

#include "sniffer.click"
ds_sniffer :: Sniffer(mazu_ds);
syn_sniffer :: Sniffer(mazu_syn);
tcp_sniffer :: Sniffer(mazu_tcp);
ntcp_sniffer :: Sniffer(mazu_ntcp);

#include "synkill.click"
#ifdef MONITOR
tcpsynkill :: SYNKill(true);
#else
tcpsynkill :: SYNKill(false);
#endif


//
// discards suspicious packets
//

#include "ds.click"
ds :: DetectSuspicious(01);

from_world -> ds;
ds [0] -> is_tcp_to_victim :: IPClassifier(tcp, -);
```

```
#ifdef MONITOR
ds [1] -> ds_split :: RatedSampler(SUSP_SNIFF);
#else
ds [1] -> ds_split :: RatedSplitter(SUSP_SNIFF);
#endif

ds_split [1] -> ds_sniffer;
ds_split [0]
#ifdef MONITOR
  -> is_tcp_to_victim;
#else
  -> Discard;
#endif

//
// monitor TCP ratio
//

#include "monitor.click"
tcp_ratemon :: TCPTrafficMonitor;

is_tcp_to_victim [0] -> tcp_monitor :: TCPMonitor -> [0] tcp_ratemon;
from_victim -> is_tcp_to_world :: IPClassifier(tcp, -);
is_tcp_to_world [0] -> [1] tcp_ratemon;

//
// enforce correct TCP ratio
//

check_tcp_ratio :: RatioShaper(1,2,40,0.2);
tcp_ratemon [0] -> check_tcp_ratio;

#ifdef MONITOR
check_tcp_ratio [1] -> tcp_split :: RatedSampler(TCP_SNIFF);
#else
check_tcp_ratio [1] -> tcp_split :: RatedSplitter(TCP_SNIFF);
#endif

tcp_split [1] -> tcp_sniffer;
tcp_split [0]
#ifdef MONITOR
  -> [0] tcpsynkill;
#else
  -> Discard;
#endif
```

```
//
// prevent SYN bomb
//

check_tcp_ratio [0] -> [0] tcpsynkill;
tcp_ratemon [1] -> [1] tcpsynkill;

tcpsynkill [0] -> to_victim_s1;
tcpsynkill [1] -> to_world;

tcpsynkill [2]
#ifdef MONITOR
  -> syn_sniffer;
Idle -> to_victim_prio;
#else
  -> tcpsynkill_split :: Tee(2)
tcpsynkill_split [0] -> to_victim_prio;
tcpsynkill_split [1] -> syn_sniffer;
#endif

//
// monitor all non TCP traffic
//

ntcp_ratemon :: IPRateMonitor(PACKETS, 0, 1, 100, 4096, false);
is_tcp_to_victim [1] -> ntcp_monitor :: NonTCPMonitor -> ntcp_t :: Tee(2);
ntcp_t [0] -> [0] ntcp_ratemon [0] -> Discard;
ntcp_t [1] -> [1] ntcp_ratemon;

//
// rate limit ICMP traffic
//

ntcp_ratemon [1] -> is_icmp :: IPClassifier(icmp, -);
is_icmp [0] -> icmp_split :: RatedSplitter (ICMP_RATE);

icmp_split [1] -> to_victim_s2;
icmp_split [0] -> icmp_sample :: RatedSampler (ICMP_SNIFF);

icmp_sample [1] -> ntcp_sniffer;
icmp_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif
```

```
//
// rate limit other non TCP traffic (mostly UDP)
//

is_icmp [1] -> is_frag :: Classifier(6/0000, -);

is_frag [0] -> udp_split :: RatedSplitter (UDP_NF_RATE);

udp_split [0] -> udp_sample :: RatedSampler (UDP_NF_SNIFF);
udp_sample [1] -> ntcp_sniffer;
udp_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif

is_frag [1] -> udp_f_split :: RatedSplitter (UDP_F_RATE);

udp_f_split [0] -> udp_f_sample :: RatedSampler (UDP_F_SNIFF);
udp_f_sample [1] -> ntcp_sniffer;
udp_f_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif

//
// further shape non-TCP traffic with ICMP dest unreachable packets
//

is_tcp_to_world [1] -> is_icmp_unreach :: IPClassifier(icmp type 3, -);
is_icmp_unreach [1] -> to_world;
is_icmp_unreach [0]
  -> icmp_unreach_counter :: Counter;

#ifndef MONITOR

icmp_unreach_counter -> icmperr_sample :: RatedSampler (UNREACH_SNIFF);
icmperr_sample [1] -> ntcp_sniffer;
icmperr_catcher :: AdaptiveShaper(.1, 50);
udp_split [1] -> [0] icmperr_catcher [0] -> to_victim_s2;
udp_f_split [1] -> [0] icmperr_catcher;
icmperr_sample [0] -> [1] icmperr_catcher [1] -> to_world;
```

```
#else

udp_split [1] -> to_victim_s2;
udp_f_split [1] -> to_victim_s2;
icmp_unreach_counter [0] -> to_world;

#endif
```

== if.click

```
//
// input/output ethernet interface for router
//
// this configuration file leaves the following elements to be hooked up:
//
// from_victim:   packets coming from victim
// from_world:    packets coming from world
// to_world:      packets going to world
// to_victim_prio: high priority packets going to victim
// to_victim_s1:  best effort packets going to victim, tickets = 4
// to_victim_s2:  best effort packets going to victim, tickets = 1
//
// see bridge.click for a simple example of how to use this configuration.

// victim network is 1.0.0.0/8 (eth1, 00:C0:95:E2:A8:A0)
//  world network is 2.0.0.0/8 (eth2, 00:C0:95:E2:A8:A1) and
//                   3.0.0.0/8 (eth3, 00:C0:95:E1:B5:38)

// ethernet input/output, forwarding, and arp machinery

tol :: ToLinux;
t :: Tee(6);
t[5] -> tol;

arpq1_prio :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
arpq1_s1 :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
arpq1_s2 :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
ar1 :: ARPResponder(1.0.0.1/32 00:C0:95:E2:A8:A0);
arpq2 :: ARPQuerier(2.0.0.1, 00:C0:95:E2:A8:A1);
ar2 :: ARPResponder(2.0.0.1/32 00:C0:95:E2:A8:A1);
arpq3 :: ARPQuerier(3.0.0.1, 00:C0:95:E1:B5:38);
ar3 :: ARPResponder(3.0.0.1/32 00:C0:95:E1:B5:38);
```

```
psched :: PrioSched;
ssched :: StrideSched (4,1);

out1_s1 :: Queue(256) -> [0] ssched;
out1_s2 :: Queue(256) -> [1] ssched;
out1_prio :: Queue(256) -> [0] psched;
ssched -> [1] psched;
psched[0] -> to_victim_counter :: Counter -> todev1 :: ToDevice(eth1);

out2 :: Queue(1024) -> todev2 :: ToDevice(eth2);
out3 :: Queue(1024) -> todev3 :: ToDevice(eth3);

to_victim_prio :: Counter -> tvpc :: Classifier(16/01, -);
tvpc [0] -> [0]arpq1_prio -> out1_prio;
tvpc [1] -> Discard;

to_victim_s1 :: Counter -> tvs1c :: Classifier(16/01, -);
tvs1c [0] -> [0]arpq1_s1 -> out1_s1;
tvs1c [1] -> Discard;

to_victim_s2 :: Counter -> tvs2c :: Classifier(16/01, -);
tvs2c [0] -> [0]arpq1_s2 -> out1_s2;
tvs2c [1] -> Discard;

to_world :: Counter -> twc :: Classifier(16/02, 16/03, -);
twc [0] -> [0]arpq2 -> out2;
twc [1] -> [0]arpq3 -> out3;
twc [2] -> Discard;

from_victim :: GetIPAddress(16);
from_world :: GetIPAddress(16);

indev1 :: PollDevice(eth1);
c1 :: Classifier (12/0806 20/0001,
             12/0806 20/0002,
                 12/0800,
                 -);
indev1 -> from_victim_counter :: Counter -> c1;
c1 [0] -> ar1 -> out1_s1;
c1 [1] -> t;
c1 [2] -> Strip(14) -> MarkIPHeader -> from_victim;
c1 [3] -> Discard;
t[0] -> [1] arpq1_prio;
t[1] -> [1] arpq1_s1;
t[2] -> [1] arpq1_s2;
```

```
indev2 :: PollDevice(eth2);
c2 :: Classifier (12/0806 20/0001,
                  12/0806 20/0002,
                      12/0800,
                      -);
indev2 -> from_attackers_counter :: Counter -> c2;
c2 [0] -> ar2 -> out2;
c2 [1] -> t;
c2 [2] -> Strip(14) -> MarkIPHeader -> from_world;
c2 [3] -> Discard;
t[3] -> [1] arpq2;

indev3 :: PollDevice(eth3);
c3 :: Classifier (12/0806 20/0001,
                  12/0806 20/0002,
                      12/0800,
                      -);
indev3 -> c3;
c3 [0] -> ar3 -> out3;
c3 [1] -> t;
c3 [2] -> Strip(14) -> MarkIPHeader -> from_world;
c3 [3] -> Discard;
t[4] -> [1] arpq3;

ScheduleInfo(todev1 10, indev1 1,
        todev2 10, indev2 1,
          todev3 10, indev3 1);
```

```
== sampler.click
```

```
elementclass RatedSampler {
 $rate |
  input -> s :: RatedSplitter($rate);
  s [0] -> [0] output;
  s [1] -> t :: Tee;
  t [0] -> [0] output;
  t [1] -> [1] output;
};

elementclass ProbSampler {
 $prob |
  input -> s :: ProbSplitter($prob);
  s [0] -> [0] output;
  s [1] -> t :: Tee;
  t [0] -> [0] output;
  t [1] -> [1] output;
};
```

== sniffer.click

```
// setup a sniffer device, with a testing IP network address
//
// argument: name of the device to setup and send packet to

elementclass Sniffer {
 $dev |
  FromLinux($dev, 192.0.2.0/24) -> Discard;

  input -> sniffer_ctr :: Counter
      -> ToLinuxSniffers($dev);
};

// note: ToLinuxSniffers take 2 us
```

== synkill.click

```
//
// SYNKill
//
// argument: true if monitor only, false if defend
//
// expects: input 0 - TCP packets with IP header to victim network
//          input 1 - TCP packets with IP header to rest of internet
//
// action: protects against SYN flood by prematurely finishing the three way
//          handshake protocol.
//
// outputs: output 0 - TCP packets to victim network
//          output 1 - TCP packets to rest of internet
//          output 2 - control packets (created by TCPSYNProxy) to victim
//

elementclass SYNKill {
 $monitor |
  // TCPSYNProxy(MAX_CONNS, THRESH, MIN_TIMEOUT, MAX_TIMEOUT,
PASSIVE);
  tcpsynproxy :: TCPSYNProxy(128, 4, 8, 80, $monitor);
 input [0] -> [0] tcpsynproxy [0] -> [0] output;
 input [1] -> [1] tcpsynproxy [1] -> [1] output;
 tcpsynproxy [2]
   -> GetIPAddress(16)
   -> [2] output;
};
```

== ds.click

```
//
// DetectSuspicious
//
// argument: takes in the victim network address and mask. for example:
//    DetectSuspicious(121A0400%FFFFFF00)
//
// expects: IP packets.
//
// action: detects packets with bad source addresses;
//        detects direct broadcast packets;
//        detects ICMP redirects.
//
// outputs: output 0 push out accepted packets, unmodified;
//         output 1 push out rejected packets, unmodified.
//

elementclass DetectSuspicious {
 $vnet |

 // see http://www.ietf.org/internet-drafts/draft-manning-dsua-03.txt for a
 // list of bad source addresses to block out. we also block out packets with
 // broadcast dst addresses.

 bad_addr_filter :: Classifier(
   12/$vnet,               // port  0: victim network address
   12/00,                  // port  1: 0.0.0.0/8 (special purpose)
   12/7F,                  // port  2: 127.0.0.0/8 (loopback)
   12/0A,                  // port  3: 10.0.0.0/8 (private network)
   12/AC10%FFF0,           // port  4: 172.16.0.0/12 (private network)
   12/C0A8,                // port  5: 192.168.0.0/16 (private network)
   12/A9FE,                // port  6: 169.254.0.0/16 (autoconf addr)
   12/C0000200%FFFFFF00,   // port  7: 192.0.2.0/24 (testing addr)
   12/E0%F0,               // port  8: 224.0.0.0/4 (class D - multicast)
   12/F0%F0,               // port  9: 240.0.0.0/4 (class E - reserved)
   12/00FFFFFF%00FFFFFF,   // port 10: broadcast saddr X.255.255.255
```

```
  12/0000FFFF%0000FFFF,      // port 11: broadcast saddr X.Y.255.255
  12/000000FF%000000FF,      // port 12: broadcast saddr X.Y.Z.255
  16/00FFFFFF%00FFFFFF,      // port 13: broadcast daddr X.255.255.255
  16/0000FFFF%0000FFFF,      // port 14: broadcast daddr X.Y.255.255
  16/000000FF%000000FF,      // port 15: broadcast daddr X.Y.Z.255
  9/01,                 // port 16: ICMP packets
  -);

input -> bad_addr_filter;
bad_addr_filter [0]  -> [1] output;
bad_addr_filter [1]  -> [1] output;
bad_addr_filter [2]  -> [1] output;
bad_addr_filter [3]  -> [1] output;
bad_addr_filter [4]  -> [1] output;
bad_addr_filter [5]  -> [1] output;
bad_addr_filter [6]  -> [1] output;
bad_addr_filter [7]  -> [1] output;
bad_addr_filter [8]  -> [1] output;
bad_addr_filter [9]  -> [1] output;
bad_addr_filter [10] -> [1] output;
bad_addr_filter [11] -> [1] output;
bad_addr_filter [12] -> [1] output;
bad_addr_filter [13] -> [1] output;
bad_addr_filter [14] -> [1] output;
bad_addr_filter [15] -> [1] output;

// ICMP rules: drop all fragmented and redirect ICMP packets

bad_addr_filter [16]
  -> is_icmp_frag_packets :: Classifier(6/0000, -);
is_icmp_frag_packets [1] -> [1] output;

is_icmp_frag_packets [0]
  -> is_icmp_redirect :: IPClassifier(icmp type 5, -);
is_icmp_redirect [0] -> [1] output;

// finally, allow dynamic filtering of bad src addresses we discovered
// elsewhere in our script.

dyn_saddr_filter :: AddrFilter(SRC, 32);
is_icmp_redirect [1] -> dyn_saddr_filter;
bad_addr_filter [17] -> dyn_saddr_filter;
dyn_saddr_filter [0] -> [0] output;
dyn_saddr_filter [1] -> [1] output;

};
```

== monitor.click

```
//
// TCPTrafficMonitor
//
// expects: input 0 takes TCP packets w IP header for the victim network;
//         input 1 takes TCP packets w IP Header from the victim network.
// action: monitors packets passing by
// outputs: output 0 - packets for victim network, unmodified;
//         output 1 - packets from victim network, unmodified.
//

elementclass TCPTrafficMonitor {
  // fwd annotation = rate of src_addr, rev annotation = rate of dst_addr
  tcp_rm :: IPRateMonitor(PACKETS, 0, 1, 100, 4096, true);

  // monitor all TCP traffic to victim, monitor non-RST packets from victim
  input [0] -> [0] tcp_rm [0] -> [0] output;
  input [1] -> i1_tcp_rst :: IPClassifier(rst, -);
  i1_tcp_rst[0] -> [1] output;
  i1_tcp_rst[1] -> [1] tcp_rm [1] -> [1] output;
};
```

Appendix listing of additional Click modules ("elements").

```
ADAPTIVESHAPER(n)                               ADAPTIVESHAPER(n)


NAME
       AdaptiveShaper - Click element

SYNOPSIS
       AdaptiveShaper(DROP_P, REPRESS_WEIGHT)

PROCESSING TYPE
       Push

DESCRIPTION
       AdaptiveShaper is a push element that shapes input traffic
       from input port 0 to output port 0. Packets are shaped
       based on "repressive" traffic from input port 1 to output
       port 1. Each repressive packet increases a multiplicative
       factor f by REPRESS_WEIGHT. Each input packet is killed
       instead of pushed out with f * DROP_P probability. After
       each dropped packet, f is decremented by 1.


EXAMPLES
ELEMENT HANDLERS
       drop_prob (read/write)
            value of DROP_P



       repress_weight (read/write)
            value of REPRESS_WEIGHT




SEE ALSO
       PacketShaper(n), RatioShaper(n)
```

ADAPTIVESPLITTER(n) ADAPTIVESPLITTER(n)

NAME

AdaptiveSplitter - Click element

SYNOPSIS

AdaptiveSplitter(RATE)

PROCESSING TYPE

Push

DESCRIPTION

AdaptiveSplitter attempts to split RATE number of packets per second for each address. It takes the fwd_rate annotation set by IPRateMonitor(n), and calculates a split probability based on that rate. The split probability attempts to guarantee RATE number of packets per second. That is, the lower the fwd_rate, the higher the split probability.

Splitted packets are on output port 1. Other packets are on output port 0.

EXAMPLES

AdaptiveSplitter(10);

SEE ALSO

IPRateMonitor(n)

ADDRFILTER(n) ADDRFILTER(n)

NAME

AddrFilter - Click element

SYNOPSIS

AddrFilter(DST/SRC, N)

PROCESSING TYPE

Push

DESCRIPTION

Filters out IP addresses given in write handler. DST/SRC specifies which IP address (dst or src) to filter. N is the maximum number of IP addresses to filter at any time. Packets passed the filter goes to output 0. Packets rejected by the filter goes to output 1.

AddrFilter looks at addresses in the IP header of the packet, not the annotation. It requires an IP header annotation (MarkIPHeader(n)).

EXAMPLES

AddrFilter(DST, 8)

Filters by dst IP address, up to 8 addresses.

ELEMENT HANDLERS

table ((read))

Dumps the list of addresses to filter and

add ((write))

Expects a string "addr mask duration", where addr is an IP address, mask is a netmask, and duration is the number of seconds to filter packets from this IP address. If 0 is given as a duration, filtering is removed. For example, "18.26.4.0 255.255.255.0 10" would filter out all packets with dst or source address 18.26.4.* for 10 seconds. New addresses push out old addresses if more than N number of filters already exist.

reset ((write))

Resets on write.

SEE ALSO

Classifier(n), MarkIPHeader(n)

ATTACKLOG(n) ATTACKLOG(n)

NAME

AttackLog - Click element; maintains a log of attack packets in SAVE_FILE.

SYNOPSIS

AttackLog(SAVE_FILE, INDEX_FILE, MULTIPLIER, PERIOD)

PROCESSING TYPE

Agnostic

DESCRIPTION

Maintains a log of attack packets in SAVE_FILE. Expects packets with ethernet headers, but with the first byte of the ethernet header replaced by an attack bitmap, set in kernel. AttackLog classifies each packet by the type of attack, and maintains an attack rate for each type of attack. The attack rate is the arrival rate of attack packets multiplied by MULTIPLIER.

AttackLog writes a block of data into SAVE_FILE once every PERIOD number of seconds. Each block is composed of entries of the following format:

delimiter (0s)	4 bytes
time	4 bytes
attack type	2 bytes
attack rate	4 bytes
ip header and payload (padded)	86 bytes
	100 bytes

Entries with the same attack type are written out together. A delimiter of 0xFFFFFFFF is written to the end of each block.

A circular timed index file is kept in INDEX_FILE along side the attacklog. See CircularIndex(n).

SEE ALSO

CircularIndex(n)

NAME

CircularIndex - Click element; writes a timed circular index into a file.

SYNOPSIS

CircularIndex

DESCRIPTION

CircularIndex writes an entry into a circular index file periodically. The entry contains a 32 bit time stamp and a 64 bit offset into another file. The following functions are exported by CircularIndex.

int initialize(String FILE, unsigned PERIOD, unsigned WRAP) - Use FILE as the name of the circular file. Writes entry into circular file once every PERIOD number of seconds. WRAP is the number of writes before wrap around. If WRAP is 0, the file is never wrapped around.

void write_entry(long long offset) - Write entry into index file. Use offset as the offset in the entry.

SEE ALSO

GatherRates(n), MonitorSRC16(n)

NAME

DiscardToDevice - Click element; drops all packets. gives skbs to device.

SYNOPSIS

DiscardToDevice(DEVICE)

PROCESSING TYPE

Agnostic

DESCRIPTION

Discards all packets received on its single input. Gives all skbuffs to specified device.

```
FILTERTCP(n)                                                FILTERTCP(n)

NAME
       FilterTCP - Click element

SYNOPSIS
       FilterTCP()

PROCESSING TYPE
       Push

DESCRIPTION
       Expects TCP/IP packets as input.
```

NAME

FromTunnel - Click element

SYNOPSIS

FromTunnel(TUNNEL, SIZE, BURST)

PROCESSING TYPE

Push

DESCRIPTION

Grab packets from kernel KUTunnel element. TUNNEL is a /proc file in the handler directory of the KUTunnel element. SIZE specifies size of the buffer to use (if packet in kernel has larger size, it is dropped). BURST specifies the maximum number of packets to push each time FromTunnel runs.

EXAMPLES

```
FromTunnel(/proc/click/tunnel/config)
```

GATHERRATES(n) GATHERRATES(n)

NAME

GatherRates - Click element

SYNOPSIS

GatherRates(SAVE_FILE, INDEX_FILE, TCPMONITOR_IN, TCPMONITOR_OUT, MONITOR_PERIOD, SAVE_PERIOD);

PROCESSING TYPE

Agnostic

DESCRIPTION

Gathers aggregate traffic rates from TCPMonitor(n) element at TCPMONITOR_IN and TCPMONITOR_OUT.

Aggregate rates are gathered once every MONITOR_PERIOD number of seconds. They are averaged and saved to SAVE_FILE once every SAVE_PERIOD number of seconds. The following entry is written to SAVE_FILE for both incoming and outgoing traffic:

delimiter (0s)	4 bytes
time	4 bytes
type (0 for incoming traffic, 1 for outgoing traffic)	4 bytes
packet rate of tcp traffic	4 bytes
byte rate of tcp traffic	4 bytes
rate of fragmented tcp packets	4 bytes
rate of tcp syn packets	4 bytes
rate of tcp fin packets	4 bytes
rate of tcp ack packets	4 bytes
rate of tcp rst packets	4 bytes
rate of tcp psh packets	4 bytes
rate of tcp urg packets	4 bytes
packet rate of non-tcp traffic	4 bytes
byte rate of non-tcp traffic	4 bytes
rate of fragmented non-tcp traffic	4 bytes
rate of udp packets	4 bytes
rate of icmp packets	4 bytes
rate of all other packets	4 bytes
---	
	72 bytes

After the two entries, an additional delimiter of 0xFFFFFFFF is written. SAVE_PERIOD must be a multiple of MONITOR_PERIOD.

A circular timed index is kept along side the stats file. See CircularIndex(n).

SEE ALSO

TCPMonitor(n) CircularIndex(n)

ICMPPINGENCAP(n) ICMPPINGENCAP(n)

NAME

ICMPPINGEncap - Click element

SYNOPSIS

ICMPPINGEncap(SADDR, DADDR [, CHECKSUM?])

DESCRIPTION

Encapsulates each incoming packet in a ICMP ECHO/IP packet with source address SADDR and destination address DADDR. The ICMP and IP checksums are calculated if CHECKSUM? is true; it is true by default.

EXAMPLES

ICMPPINGEncap(1.0.0.1, 2.0.0.2)

```
KUTUNNEL(n)                                                      KUTUNNEL(n)

NAME
       KUTunnel  -  Click element; stores packets in a FIFO queue
       that userlevel Click elements pull from.

SYNOPSIS
       KUTunnel([CAPACITY])

PROCESSING TYPE
       Push

DESCRIPTION
       Stores incoming packets  in  a  first-in-first-out  queue.
       Drops incoming packets if the queue already holds CAPACITY
       packets. The default for CAPACITY is  1000.  Allows  user-
       level elements to pull from queue via ioctl.

ELEMENT HANDLERS
       length (read-only)
            Returns the current number of packets in the queue.


       highwater_length (read-only)
            Returns  the maximum number of packets that have ever
            been in the queue at once.


       capacity (read/write)
            Returns or sets the queue's capacity.


       drops (read-only)
            Returns the number of packets dropped so far.



SEE ALSO
       Queue(n)
```

NAME

Logger - Click element

SYNOPSIS

Logger(LOGFILE, INDEXFILE [, LOCKFILE, COMPRESS?, LOGSIZE, PACKETSIZE, WRITEPERIOD, IDXCOALESC, PACKETFREQ, MAXBUFSIZE])

PROCESSING TYPE

Agnostic

DESCRIPTION

Has one input and one output.

Write packets to log file LOGFILE. A log file is a circular buffer containing packet records of the following form:

```
 --------------------
|   time (6 bytes)   |
|  length (2 bytes)  |
|    packet data     |
 --------------------
```

Time is the number of seconds and milliseconds since the Epoch at which a given packet was seen. Length is the length (in bytes) of the subsequent logged packet data. One or more packet records constitute one packet sequence.

INDEXFILE maintains control data for LOGFILE. It contains a sequence of sequence control blocks of the following form:

```
 -----------------------
|    date (4 bytes)     |
| offset (sizeof off_t) |
| length (sizeof off_t) |
 -----------------------
```

Date is a number of seconds since the Epoch. Offset points to the beginning of the packet sequence, i.e. to the earliest packet record having a time no earlier than date. Length is the number of bytes in the packet sequence. IDXCOALESC is the number of coalescing packets that a control block always cover. Default is 1024.

Sequence control blocks are always stored in increasing chronological order; offsets need not be in increasing order, since LOGFILE is a circular buffer.

COMPRESS? (true, false) determines whether packet data is logged in compressed form. Default is true.

LOGSIZE specifies the maximum allowable log file size, in KB. Default is 2GB. LOGSIZE=0 means "grow as necessary".

PACKETSIZE is the amount of packet data stored in the log. By default, the first 120 (128-6-2) bytes are logged and the remainder is discarded. Note that PACKETSIZE is the amount of data logged before compression.

Packet records are buffered in memory and periodically written to LOGFILE as a packet sequence. WRITEPERIOD is the number of seconds that should elapse between writes to LOGFILE. Default is 60. INDEXFILE is updated every time a sequence of buffered packet records is written to LOGFILE. The date in the sequence control block is the time of the first packet record of the sequence, with milliseconds omitted.

PACKETFREQ is an estimate of the number of packets per second that will be passing through Logger. Combined with WRITEPERIOD, this is a hint of buffer memory requirements. By default, PACKETFREQ is 1000. Since by default WRITEPERIOD is 60 and each packet record is at most 128 bytes, Logger normally allocates 7500KB of memory for the buffer. Logger will grow the memory buffer as needed up to a maximum of MAXBUFSIZE KB, at which point the buffered packet records are written to disk even if WRITEPERIOD seconds have not elapsed since the last write. Default MAXBUFSIZE is 65536 (64MB).

NAME

MonitorSRC16 - Click element

SYNOPSIS

MonitorSRC16(SAVE_FILE, INDEX_FILE, MULTIPLIER, PERIOD, WRAP)

PROCESSING TYPE

Agnostic

DESCRIPTION

Examines src address of packets passing by. Collects statistics for each 16 bit IP address prefix. The following data structure is written to SAVE_FILE for every 16 bit IP address prefix every PERIOD number of seconds.

delimiter (0s)	(4 bytes)
time	(4 bytes)
addr	(4 bytes)
tcp rate	(4 bytes)
non tcp rate	(4 bytes)
percent of tcp	(1 byte)
percent of tcp frag	(1 byte)
percent of tcp syn	(1 byte)
percent of tcp fin	(1 byte)
percent of tcp ack	(1 byte)
percent of tcp rst	(1 byte)
percent of tcp psh	(1 byte)
percent of tcp urg	(1 byte)
percent of non tcp frag	(1 byte)
percent of udp	(1 byte)
percent of icmp	(1 byte)
reserved	(1 byte)
	32 bytes

TCP and non TCP rates are multiplied by MULTIPLIER. An additional delimiter of 0xFFFFFFFF is written at the end of a block of entries.

WARP specifies the number of writes before wrap-around. For example, if PERIOD is 60, WARP is 5, then every 5 minutes, the stats file wrap around.

A timed circular index is maintained along side the statistics file in INDEX_FILE. See CircularIndex(n).

SEE ALSO

CircularIndex(n)

RANDOMTCPIPENCAP(n) RANDOMTCPIPENCAP(n)

NAME

RandomTCPIPEncap - Click element

SYNOPSIS

RandomTCPIPEncap(DA BITS [DP SEQN ACKN CHECKSUM SA MASK])

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a TCP/IP packet with random source address and source port, destination address DA, and control bits BITS. If BITS is -1, control bits are also generated randomly. If destination port DP, sequence number SEQN, or ack number ACKN is specified and non-zero, it is used. Otherwise, it is generated randomly for each packet. IP and TCP checksums are calculated if CHECKSUM is true; it is true by default. SEQN and ACKN should be in host order. SA and MASK are optional IP address; if they are specified, the source address is computed as ((random() & MASK) | SA).

EXAMPLES

```
RandomTCPIPEncap(1.0.0.2 4)
```

SEE ALSO

RoundRobinTCPIPEncap(n), RandomUDPIPEncap(n)

RANDOMUDPIPENCAP(n) RANDOMUDPIPENCAP(n)

NAME

RandomUDPIPEncap -- Click element

SYNOPSIS

RandomUDPIPEncap(SADDR SPORT DADDR DPORT PROB [CHECKSUM?] [, ...])

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a UDP/IP packet with source address SADDR, source port SPORT, destination address DADDR, and destination port DPORT. The UDP checksum is calculated if CHECKSUM? is true; it is true by default.

PROB gives the relative chance of this argument be used over others.

The RandomUDPIPEncap element adds both a UDP header and an IP header.

You can a maximum of 16 arguments. Each argument specifies a single UDP/IP header. The element will randomly pick one argument. The relative probabilities are determined by PROB.

The Strip(n) element can be used by the receiver to get rid of the encapsulation header.

EXAMPLES

```
RandomUDPIPEncap(1.0.0.1 1234 2.0.0.2 1234 1 1,
                 1.0.0.2 1093 2.0.0.2 1234 2 1)
```

Will send about twice as much UDP/IP packets with 1.0.0.2 as its source address than packets with 1.0.0.1 as its source address.

SEE ALSO

Strip(n), UDPIPEncap(n), RoundRobinUDPIPEncap(n)

RATEWARN(n) RATEWARN(n)

NAME

RateWarn - Click element; classifies traffic and sends out warnings when rate of traffic exceeds specified rate.

SYNOPSIS

RateWarn(RATE, WARNFREQ)

PROCESSING TYPE

Push

DESCRIPTION

RateWarn has three output ports. It monitors the rate of packet arrival on input port 0. Packets are forwarded to output port 0 if rate is below RATE. If rate exceeds RATE, it sends out a warning packet WARNFREQ number of seconds apart on output port 2 in addition to forwarding all traffic through output port 1.

SEE ALSO

PacketMeter(n)

```
RATIOSHAPER(n)                                         RATIOSHAPER(n)


NAME
       RatioShaper - Click element

SYNOPSIS
       RatioShaper(FWD_WEIGHT, REV_WEIGHT, THRESH, P)

PROCESSING TYPE
       Push

DESCRIPTION
       RatioShaper  shapes  packets  based  on  fwd_rate_anno and
       rev_rate_anno rate annotations  set  by  IPRateMonitor(n).
       If   either   annotation   is  greater  than  THRESH,  and
       FWD_WEIGHT*fwd_rate_anno >  REV_WEIGHT*rev_rate_anno,  the
       packet is moved onto output port 1 with a probability of

          min(1,
P*(fwd_rate_anno*FWD_WEIGHT)/(rev_rate_anno*REV_WEIGHT))

       FWD_WEIGHT,  REV_WEIGHT,  and  THRESH are integers. P is a
       decimal between 0 and 1. Otherwise, packet is forwarded on
       output port 0.


EXAMPLES
         RatioShaper(1, 2, 100, .2);

       if  fwd_rate_anno more than twice as big as rev_rate_anno,
       and both rates are above 100, drop packets with an initial
       probability of 20 percent.


ELEMENT HANDLERS
       fwd_weight (read/write)
            value of FWD_WEIGHT


       rev_weight (read/write)
            value of REV_WEIGHT


       thresh (read/write)
            value of THRESH


       drop_prob (read/write)
            value of P


SEE ALSO
       Block(n), IPRateMonitor(n)
```

```
REPORTACTIVITY(n)                                    REPORTACTIVITY(n)

NAME
       ReportActivity - Click element

SYNOPSIS
       ReportActivity(SAVE_FILE, IDLE)

PROCESSING TYPE
       Agnostic

DESCRIPTION
       Write  into  SAVE_FILE  a 32 bit time value followed by an
       ASCII representation of that time stamp whenever a  packet
       comes  by. If IDLE number of seconds pass by w/o a packet,
       removes the file.
```

ROUNDROBINSETIPADDRESS(n) ROUNDROBINSETIPADDRESS(n)

NAME

RoundRobinSetIPAddress - Click element

SYNOPSIS

RoundRobinSetIPAddress(ADDR [, ...])

PROCESSING TYPE

Agnostic

DESCRIPTION

Set the destination IP address annotation of each packet with an address chosen from the configuration string in round robin fashion. Does not compute checksum (use SetIPChecksum(n) or SetUDPTCPChecksum(n)) or encapsulate the packet with headers (use RoundRobinUDPIPEncap(n) or RoundRobinTCPIPEncap(n) with bogus address).

EXAMPLES

```
RoundRobinUDPIPEncap(2.0.0.2 0.0.0.0 0 0 0)
  -> RoundRobinSetIPAddress(1.0.0.2, 1.0.0.3, 1.0.0.4)
  -> StoreIPAddress(12)
  -> SetIPChecksum
  -> SetUDPTCPChecksum
```

this configuration segment places an UDP header onto each packet, with randomly generated source and destination ports. The destination IP address is 2.0.0.2, the source IP address is 1.0.0.2, or 1.0.0.3, or 1.0.0.4. Both IP and UDP checksum are computed.

SEE ALSO

RoundRobinUDPIPEncap(n), RoundRobinTCPIPEncap(n), UDPIPEncap(n) , SetIPChecksum(n), SetUDPTCPChecksum(n), SetIPAddress(n), StoreIPAddress(n)

ROUNDROBINTCPIPENCAP(n) ROUNDROBINTCPIPENCAP(n)

NAME

RoundRobinTCPIPEncap - Click element

SYNOPSIS

```
RoundRobinTCPIPEncap(SA DA BITS [SP DP SEQN ACKN CHECKSUM]
[, ...])
```

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a TCP/IP packet with source address SA, source port SP (if 0, a random one is generated for each packet), destination address DA, and destination port DP (if 0, a random one is generated for each packet), and control bits BITS. If SEQN and ACKN specified are non-zero, they are used. Otherwise, they are randomly generated for each packet. IP and TCP checksums are calculated if CHECKSUM is true; it is true by default. SEQN and ACKN should be in host order.

The RoundRobinTCPIPEncap element adds both a TCP header and an IP header.

You can give as many arguments as you'd like. Each argument specifies a single TCP/IP header. The element will cycle through the headers in round-robin order.

The Strip(n) element can be used by the receiver to get rid of the encapsulation header.

EXAMPLES

```
    RoundRobinTCPIPEncap(2.0.0.2 1.0.0.2 4 1022 1234 42387492
2394839 1,
                              2.0.0.2 1.0.0.2 2)
```

SEE ALSO

Strip(n), RoundRobinUDPIPEncap(n)

ROUNDROBINUDPIPENCAP(n) ROUNDROBINUDPIPENCAP(n)

NAME

RoundRobinUDPIPEncap - Click element

SYNOPSIS

```
RoundRobinUDPIPEncap(SADDR  DADDR  [SPORT DPORT CHECKSUM?]
[, ...])
```

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a UDP/IP packet with source address SADDR, source port SPORT, destination address DADDR, and destination port DPORT. The UDP and IP checksums are calculated if CHECKSUM? is true; it is true by default. If either DPORT or SPORT is 0, the port will be randomly generated for each packet.

The RoundRobinUDPIPEncap element adds both a UDP header and an IP header.

You can give as many arguments as you'd like. Each argument specifies a single UDP/IP header. The element will cycle through the headers in round-robin order.

The Strip(n) element can be used by the receiver to get rid of the encapsulation header.

EXAMPLES

```
RoundRobinUDPIPEncap(2.0.0.2 1.0.0.2 1234 1002 1,
                     2.0.0.2 1.0.0.2 1234)
```

SEE ALSO

Strip(n), UDPIPEncap(n)

NAME

SetSniffFlags - Click element; sets sniff flags annotation.

SYNOPSIS

SetSniffFlags(FLAGS [, CLEAR])

PROCESSING TYPE

Agnostic

DESCRIPTION

Set the sniff flags annotation of incoming packets to FLAGS bitwise or with the old flags. if CLEAR is true (false by default), the old flags are ignored.

NAME

SetUDPTCPChecksum - Click element

SYNOPSIS

SetUDPTCPChecksum()

PROCESSING TYPE

Agnostic

DESCRIPTION

Expects an IP packet as input. Calculates the ICMP, UDP or TCP header's checksum and sets the checksum header field. Does not modify packet if it is not an ICMP, UDP, or TCP packet.

SEE ALSO

SetIPChecksum(n)

NAME
StoreSniffFlags - Click element; stores sniff flags annotation in packet

SYNOPSIS
StoreSniffFlags(OFFSET)

PROCESSING TYPE
Agnostic

DESCRIPTION
Copy the sniff flags annotation into the packet at offset OFFSET.

NAME
TCPMonitor - Click element

SYNOPSIS
TCPMonitor()

PROCESSING TYPE
Push

DESCRIPTION
Monitors and splits TCP traffic. Output 0 are TCP traffic, output 1 are non-TCP traffic. Keeps rates of TCP, TCP BYTE, SYN, ACK, PUSH, RST, FIN, URG, and fragmented packets. Also keeps rates of ICMP, UDP, non-TCP BYTE, and non-TCP fragmented traffic.

ELEMENT HANDLERS
rates (read)
dumps rates

NAME

TCPSYNProxy - Click element

SYNOPSIS

TCPSYNProxy(MAX_CONNS, THRESHOLD, MIN_TIMEOUT, MAX_TIMEOUT [, PASSIVE])

PROCESSING TYPE

Push

DESCRIPTION

Help settup a three way TCP handshake from A to B by supplying the last ACK packet to the SYN ACK B sent prematurely, and send RST packets to B later if no ACK was received from A.

Expects IP encapsulated TCP packets, each with its ip header marked (MarkIPHeader(n) or CheckIPHeader(n)).

Aside from responding to SYN ACK packets from B, TCPSYNProxy also examines SYN packets from A. When a SYN packet from A is received, if there are more than MAX_CONNS number of outstanding 3 way connections per destination (daddr + dport), reject the SYN packet. If MAX_CONNS is 0, no maximum is set.

The duration from sending an ACK packet to B to sending a RST packet to B decreases exponentially as the number of outstanding connections to B increases pass 2^THRESHOLD. The minimum timeout is MIN_TIMEOUT. If the number of outstanding half-open connections is above 2^THRESHOLD, the timeout is

```
max(MIN_TIMEOUT, MAX_TIMEOUT >> (N >> THRESHOLD))
```

Where N is the number of outstanding half-open connections. For example, let the MIN_TIMEOUT value be 4 seconds, the MAX_TIMEOUT value be 90 seconds, and THRESHOLD be 3. Then when N < 8, timeout is 90. When N < 16, timeout is 45. When N < 24, timeout is 22 seconds. When N < 32, timeout is 11 seconds. When N < 64, timeout is 4 seconds. Timeout period does not decrement if the threshold is 0.

TCPSYNProxy has two inputs, three outputs. All inputs and outputs take in and spew out packets with IP header. Input 0 expects TCP packets from A to B. Input 1 expects TCP packets from B to A. Output 0 spews out packets from A to B. Output 1 spews out packets from B to A. Output 2 spews out the ACK and RST packets generated by the element.

If PASSIVE is true (it is not by default), monitor TCP three-way handshake instead of actively setting it up. In this case, no ACK or RST packets will be sent. When an outstanding SYN times out, the SYN ACK packet is sent out of output port 2. No packets on port 0 are modified or dropped in this operating mode.

EXAMPLES

```
... -> CheckIPHeader() -> TCPSYNProxy(128,3,10,90) -> ...
```

ELEMENT HANDLERS

summary (read)

Returns number of ACK and RST packets sent and number of SYN packets rejected.

table (read)

Dumps the table of half-opened connections.

reset (write)

Resets on write.

SEE ALSO

MarkIPHeader(n), CheckIPHeader(n)

```
TCPSYNRESP(n)                                         TCPSYNRESP(n)

NAME
       TCPSYNResp - Click element

SYNOPSIS
       TCPSYNResp()

PROCESSING TYPE
       Push

DESCRIPTION
       Takes  in  TCP packet, if it is a SYN packet, return a SYN
       ACK. This is solely for debugging and performance  tunning
       purposes.  No checksum is done.  Spews out original packet
       on output 0 untouched. Spews out new packet on output 1.
```

What is claimed is:

1. A method of monitoring traffic flow in a monitor device disposed to receive network traffic packets comprises:

producing statistics corresponding to a parameter of traffic flow to trace the source of an attack, with producing further comprising:

mapping the traffic flow into a plurality of buckets by applying a hash function "f(h)" to the parameter of the traffic flow to output an integer corresponding to one of the buckets;

accumulating statistics from the packets; and

comparing the number of buckets to a threshold; and

determining whether the number of buckets should be divided into more buckets or combined into fewer buckets based on comparing the number of buckets to the threshold.

2. The method of claim 1 wherein the buckets are storage areas in a memory space of the monitor device.

3. The method of claim 1 wherein as the number of buckets changes, the buckets have values derived from the buckets prior to the change.

4. The method of claim 1 wherein the hash function adapts to map to the new number of buckets, as the new number of buckets changes.

5. The method of claim 1 wherein comparing statistic values comprises:

comparing the value accumulated in the bucket to a threshold that depends on the number of buckets.

6. The method of claim 1 wherein the parameter is the count of how many packets a data collector or gateway examines.

7. The method of claim 1 wherein as a value of a parameter for one bucket approaches a threshold, the monitoring device raises an alarm.

8. The method of claim 1 wherein the hash function changes periodically in a randomly secret manner so that packets are reassigned to different buckets.

9. The method of claim 1 wherein the variable number of buckets dynamically adjusts the amount of traffic and number of flows monitored, so that the monitoring device is not vulnerable to a denial of service attack against its own resources.

10. The method of claim 1 wherein the variable number of buckets efficiently identifies the source or sources of attack by breaking down traffic into different buckets and examining statistics accumulated for a parameter and a corresponding threshold in each bucket.

11. The method of claim 1 wherein the traffic is monitored at multiple levels of granularity, from aggregate to individual flows.

12. The method of claim 1 wherein the traffic is applied to monitoring of TCP packet ratios and repressor traffic.

13. The method of claim 1 wherein the threshold is a first threshold and the method further comprises:

comparing accumulated statistic values from the buckets to second threshold values to determine that an event is of significance.

14. A computer program product residing on a computer readable for monitoring network traffic flow in a network comprises instructions for causing a computer to:

map traffic flow into a plurality of buckets by applying a hash function "f(h)" to a parameter of the traffic flow to output an integer corresponding to one of the buckets;

accumulate statistics from the packets; and

compare the accumulated statistic values from the buckets to configured threshold values corresponding to the number of buckets to determine that an event is of significance; and

adjust the number of buckets as the number of buckets approaches a second threshold.

15. The computer program product of claim 14 wherein based on the second threshold, the buckets are divided into more buckets or combined into fewer buckets

16. The computer program product of claim 14 wherein instructions to monitor further comprise instructions to

divide the bucket into a different number of new buckets containing values derived from the original bucket.

17. The computer program product of claim 14 wherein the hash function adapts to map to the new number of buckets as the new number of buckets changes.

18. The computer program product of claim 14 wherein the parameter is the count of how many packets a data collector or gateway examines.

19. The computer program product of claim 14 wherein the buckets are storage areas in the memory space of the monitor device.

20. The computer program product of claim 14 wherein the hash function changes periodically in a randomly secret manner so that packets are reassigned to different buckets.

21. A data collector to collect statistical information about network flows comprises:

a computer readable medium;

a computing device that executes a computer program product stored on the computer readable medium comprising instructions to cause the computing device to:

map traffic flow into a plurality of buckets by applying a hash function "f(h)" to the parameter of the traffic flow to output an integer corresponding to one of the buckets;

accumulate statistics from the packets; and

compare the accumulated statistic values from the buckets to configured threshold values corresponding to the number of buckets to determine that an event is of significance; and adjust the number of buckets as the number of buckets approaches a second threshold; and

a port to link the data collector to a central control center.

22. A method of detecting a denial of service attack on a victim site, the method comprising:

monitoring network traffic sent to the victim site;

determining a ratio of incoming to outgoing TCP packets destined and sourced from systems at the site;

comparing the ratio to a threshold value; and

raising an alarm when the ratio exceeds the threshold value.

23. The method of claim 22 wherein monitoring further comprises:

storing the determined ratio over periods of time; and

analyzing the ratio to determine over time how much and how the ratio exceeds a value to set a threshold.

24. The method of claim 22 wherein monitoring further comprises:

establishing the ratio, and as the ratio exceeds that established threshold raising the alarm.

25. The method of claim 22 wherein the ratio is about 2:1 for incoming to outgoing TCP packets.

26. The method of claim 22 wherein as the ratio exceeds the threshold it is a indication that the victim is receiving bad TCP traffic that are not part of any established TCP connection.

27. The method of claim 22 wherein as the ratio exceeds the threshold, exceeding the threshold raises an indication that the victim is overloaded to acknowledge the requests.

28. The method of claim 22 wherein monitoring comprises:

producing statistics corresponding to the ratio by mapping the traffic flow into a plurality of buckets by applying a hash function "f(h)" to the parameter of the traffic flow to output an integer corresponding to one of the buckets.

29. The method of claim 28 further comprises:

communicating the statistics collected to a control center.

30. The method of claim 29 wherein communicating occurs over a dedicated link to the control center via a hardened network.

31. The method of claim 30 executed on a gateway physically deployed in the network.

32. A monitor device to collect statistical information about network flows comprises:

a computer readable medium;

a computing device that executes a computer program product stored on the computer readable medium comprising instructions to cause the computing device to:

monitor network traffic sent to the victim site;

determine a ratio of incoming to outgoing TCP packets destined and sourced from systems at the site;

compare the ratio to a threshold value; and

raise an alarm when the ratio exceeds the threshold value.

33. The monitor device of claim 32 wherein the monitor is a gateway.

34. The monitor device of claim 32 wherein wherein the ratio is about 2:1 for incoming to outgoing TCP packets.

35. A method of detecting a denial of service attack on a victim site, the method comprising:

monitoring network traffic sent to the victim site;

determining a ratio of incoming to outgoing TCP packets destined and sourced from systems at the site;

comparing the ratio to a threshold value; and

raising an alarm when the ratio exceeds the threshold value.

36. The method of claim 35 wherein monitoring further comprises:

storing the determined ratio over periods of time; and

analyzing the ratio to determine over time how much and how the ratio exceeds a value to set a threshold.

37. The method of claim 35 wherein monitoring further comprises:

establishing the ratio and as the ratio exceeds that threshold raising the alarm.

38. The method of claim 35 wherein the ratio is about 2:1, for incoming to outgoing TCP packets.

39. The method of claim 35 wherein the ratio is up to about 3:1, for incoming to outgoing TCP packets.

40. The method of claim 35 wherein as the ratio exceeds the threshold it is a indication that the victim is receiving bad TCP traffic that are not part of any established TCP connection.

41. The method of claim 35 wherein monitoring comprises:

producing statistics corresponding to the ratio by mapping the traffic flow into a plurality of buckets by applying a hash function "f(h)" to source addresses of the traffic flow to output an integer corresponding to one of the buckets.

42. The method of claim 41 further comprises:

communicating the statistics collected to a control center.

43. The method of claim 42 wherein communicating occurs over a dedicated link to the control center via a hardened network.

44. The method of claim 43 executed on a gateway physically deployed in the network.

45. A method of detecting a denial of service attack on a victim site, the method comprising:

monitoring network traffic sent to the victim site and identify packets generated from repressor traffic;

analyzing message header information from identified packets; and generating logs of contents of the message headers guard against attackes.

46. The method of claim 45 wherein the logs are generated for forensic purposes or to selectively block future messages similar to the ones that caused the repressor traffic messages.

47. The method of claim 45 wherein the repressor traffic is ICMP port unreachable messages that are generated by an end host when it receives a packet on a port that is not responding to requests.

48. The method of claim 45 wherein repressor traffic is any network traffic that is indicative of problems or a potential attack in a main flow of traffic.

49. The method of claim 45 wherein the method is executed on a gateway.

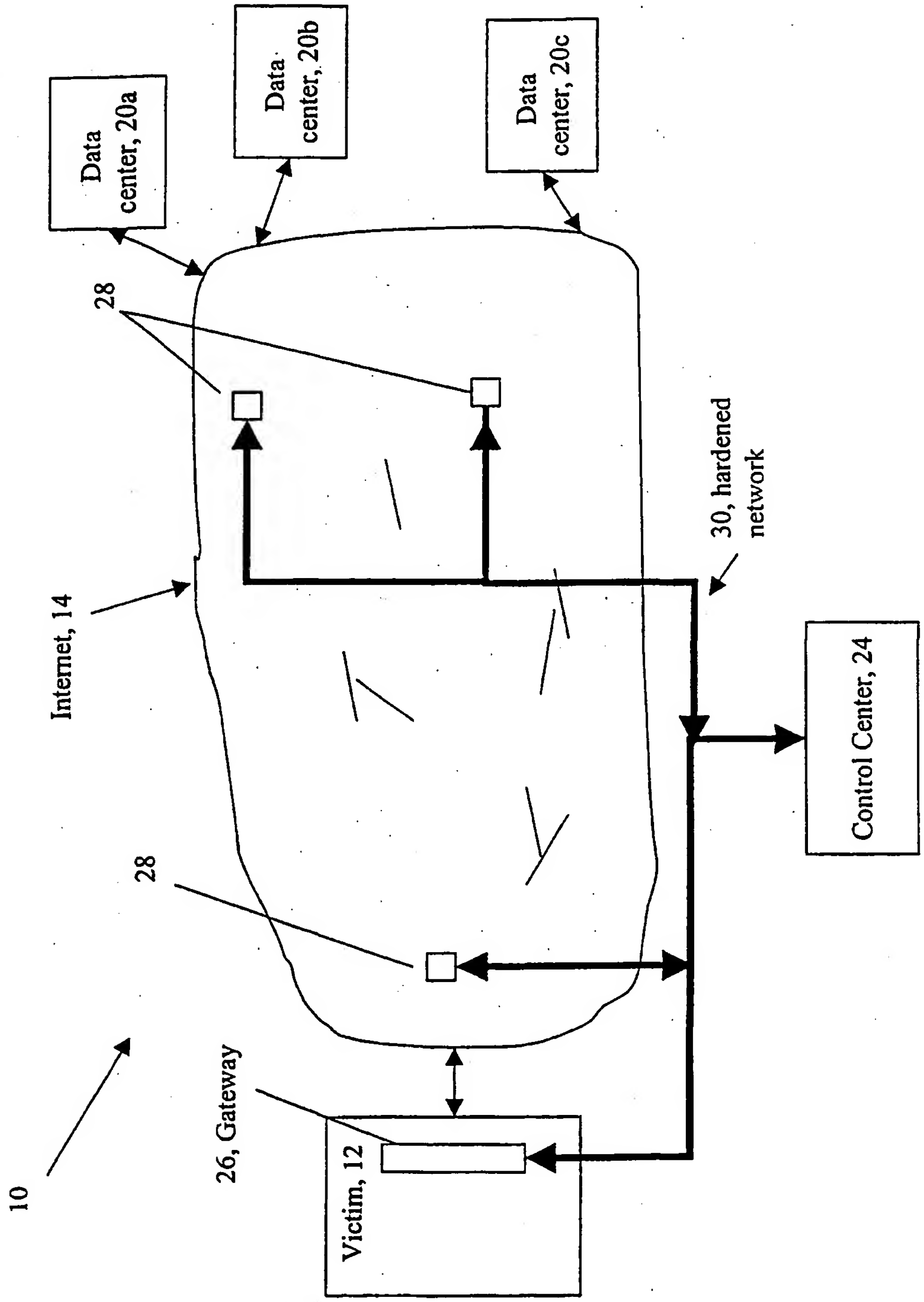

Data center, 20a
Data center, 20b
Data center, 20c
28
28
Internet, 14
30, hardened network
Control Center, 24
10
26, Gateway
Victim, 12

FIG. 1

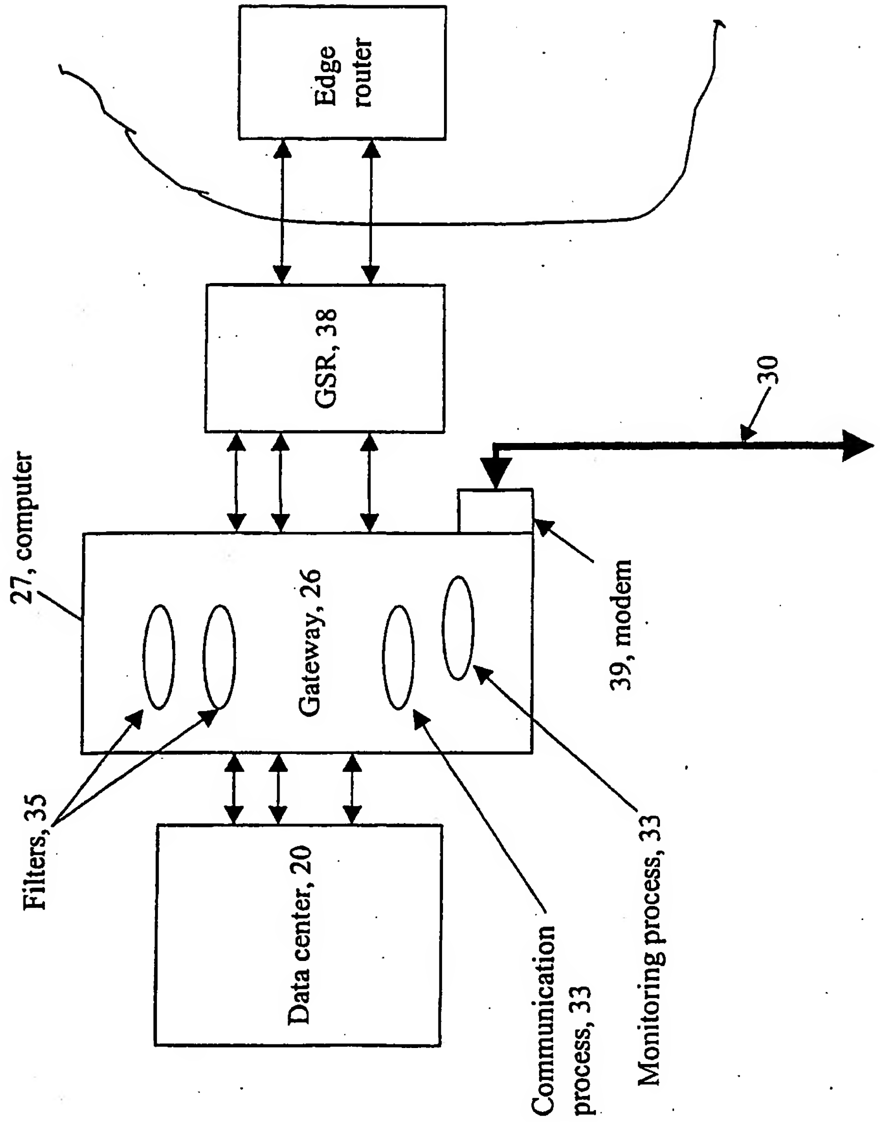

Edge router
GSR, 38
30
27, computer
Gateway, 26
39, modem
Filters, 35
Data center, 20
Communication process, 33
Monitoring process, 33

FIG. 2

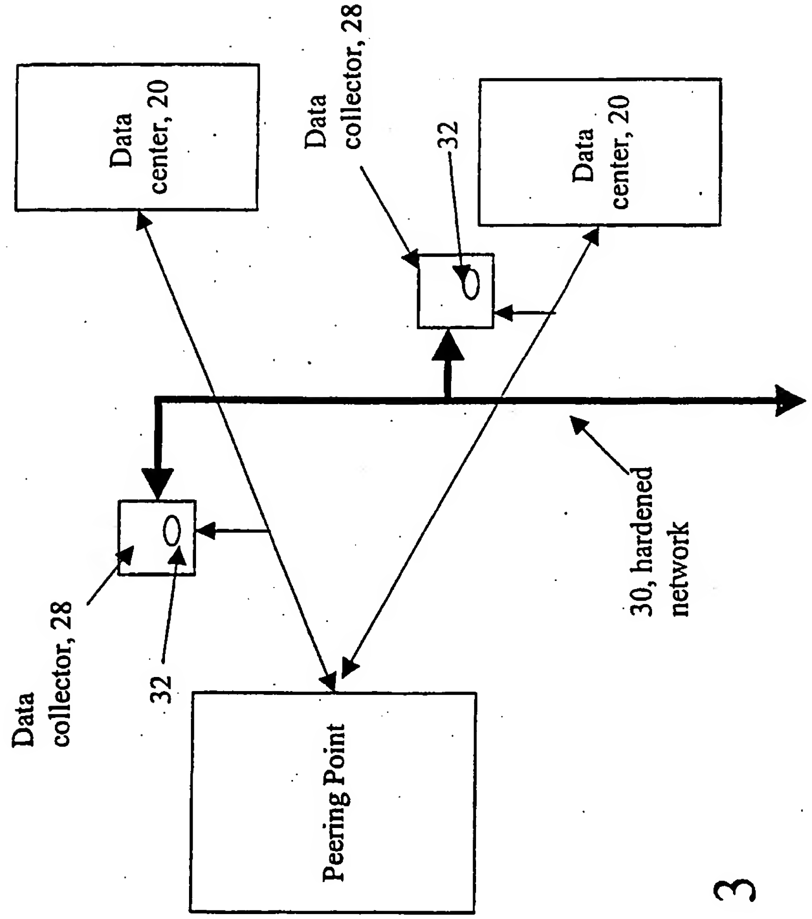

Data center, 20
Data collector, 28
32
Data center, 20
Data collector, 28
32
Peering Point
30, hardened network

FIG. 3

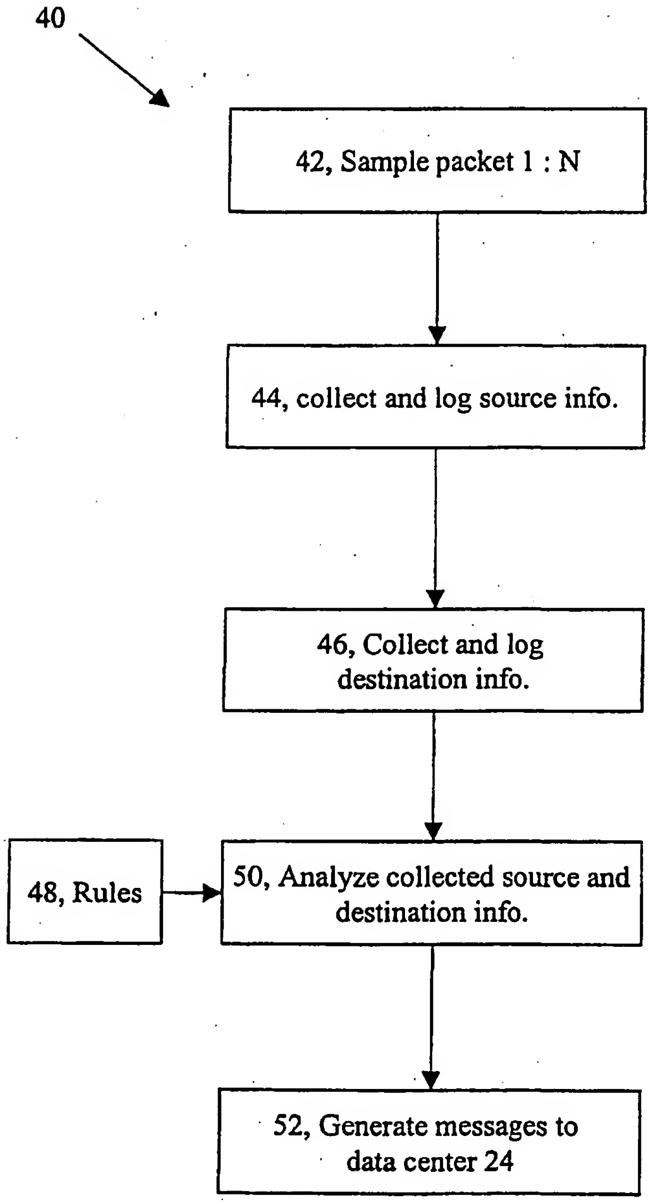

40
42, Sample packet 1 : N
44, collect and log source info.
46, Collect and log destination info.
48, Rules
50, Analyze collected source and destination info.
52, Generate messages to data center 24

FIG. 4

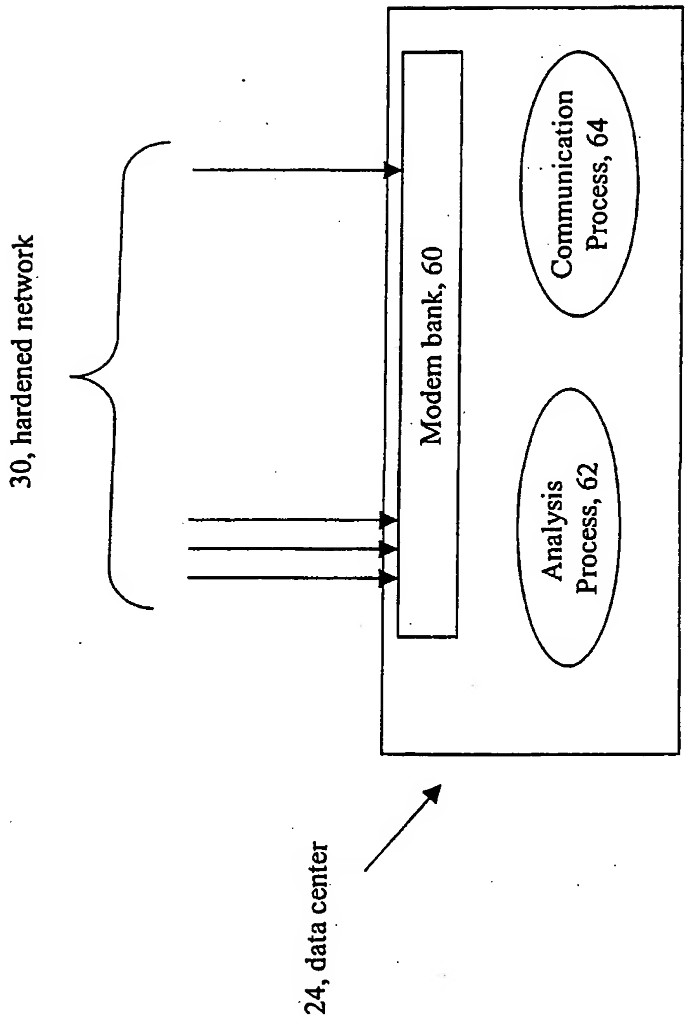

30, hardened network
Modem bank, 60
Communication Process, 64
Analysis Process, 62
24, data center

FIG. 5

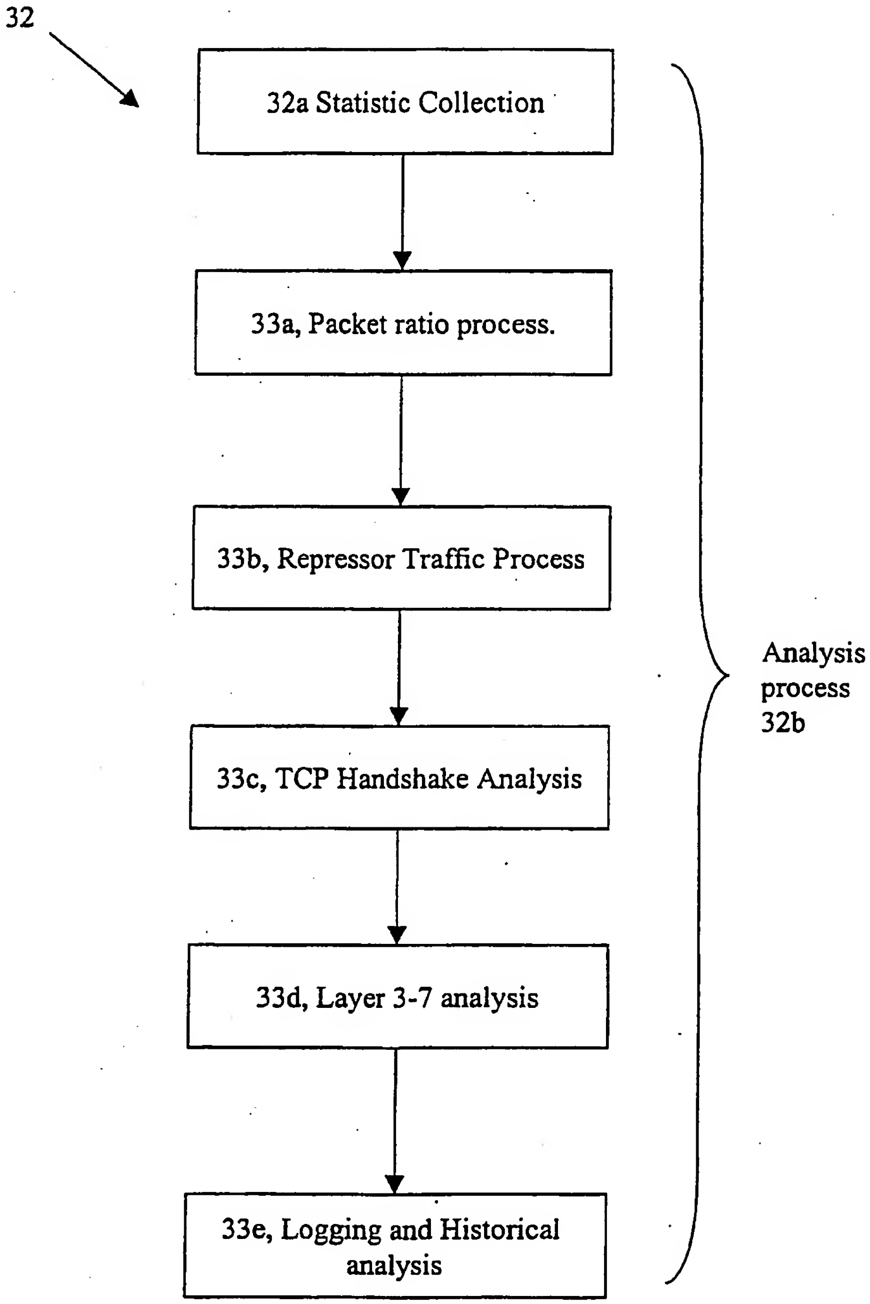

32
32a Statistic Collection
33a, Packet ratio process.
33b, Repressor Traffic Process
33c, TCP Handshake Analysis
33d, Layer 3-7 analysis
33e, Logging and Historical analysis
Analysis process 32b

FIG. 6

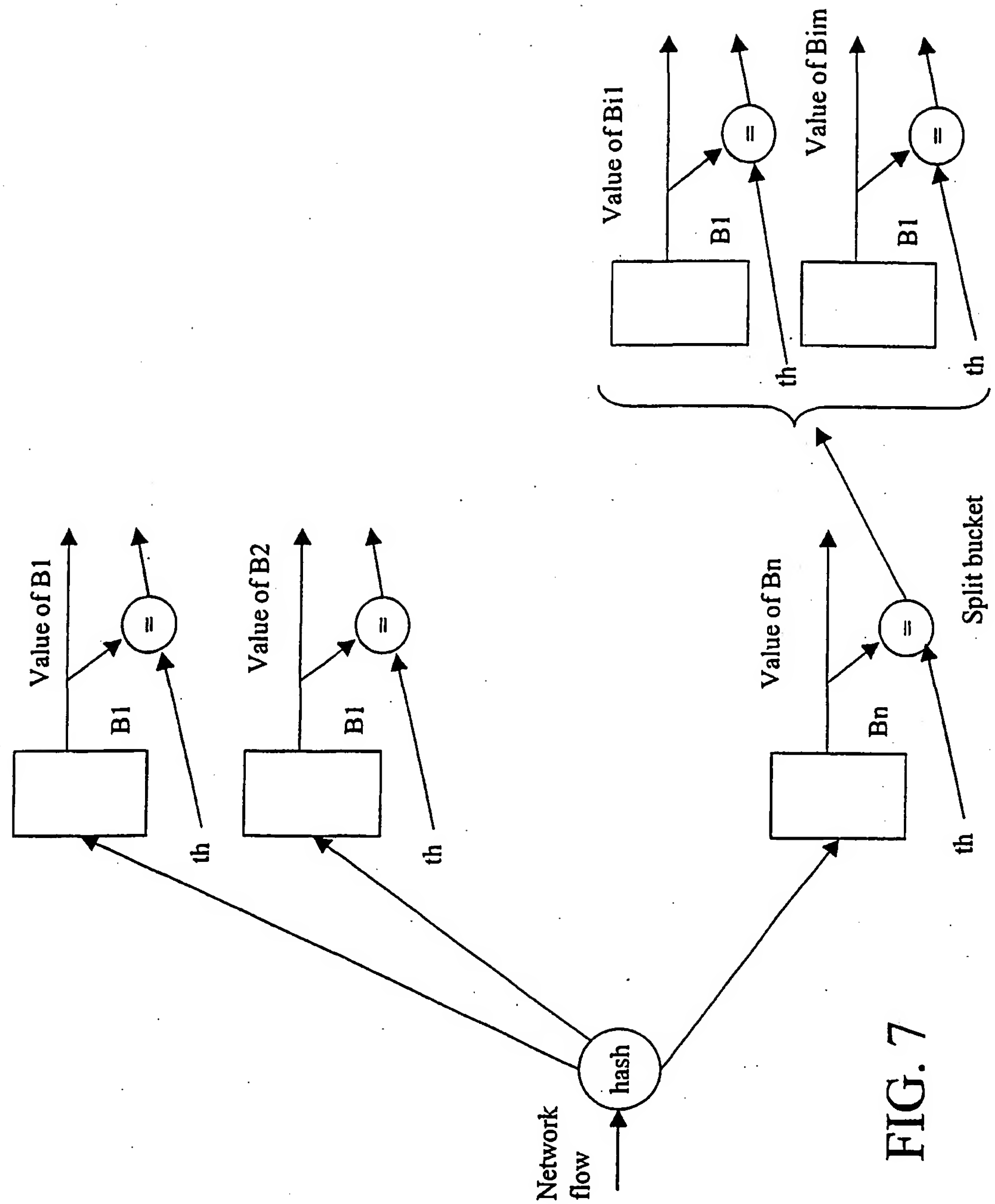
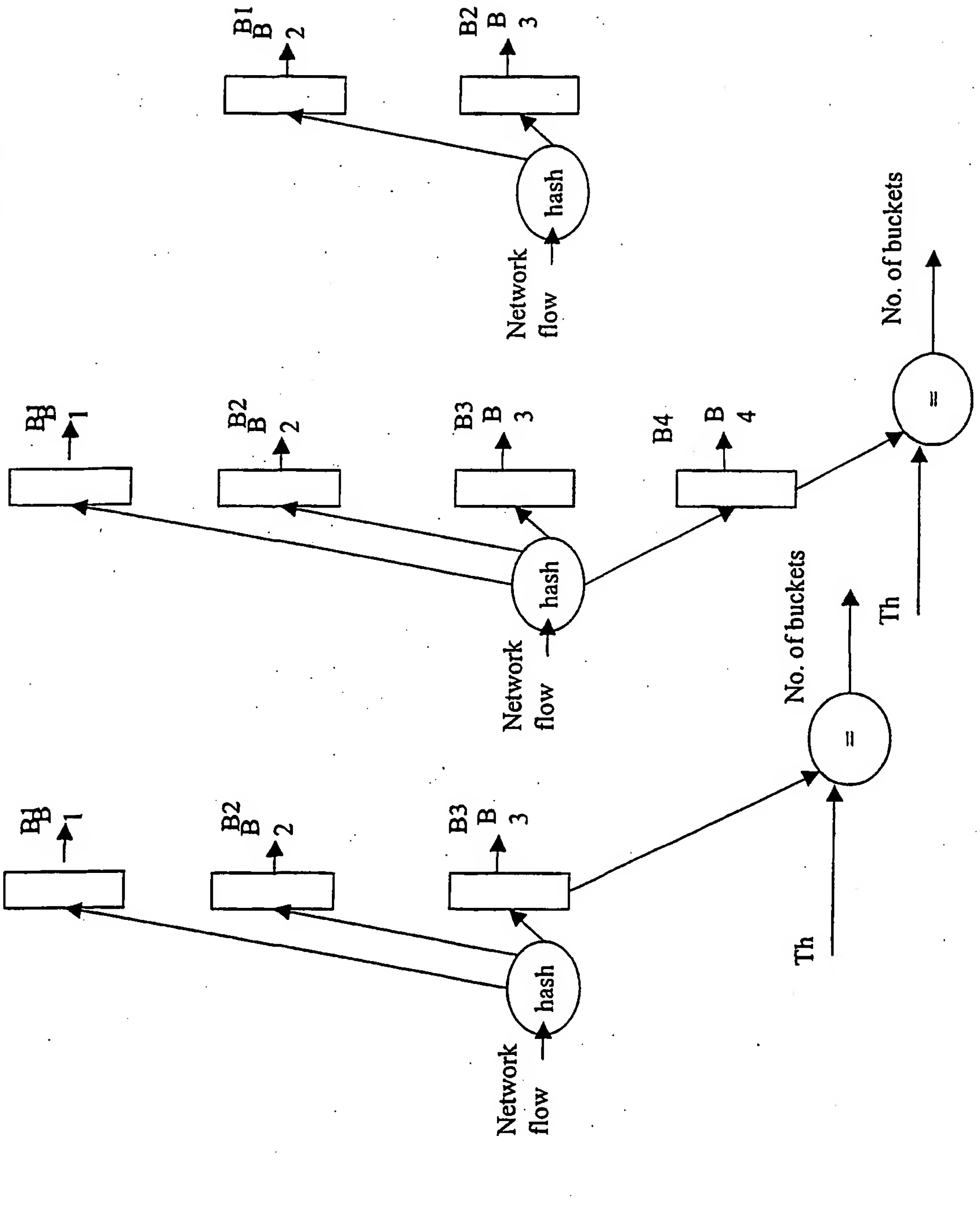

Network flow
hash
B1
Value of B1
=
th
B1
Value of B2
=
th
Bn
Value of Bn
=
th
Split bucket
B1
Value of Bi1
=
th
B1
Value of Bim
=
th

FIG. 7

FIG. 8


Network flow
hash
B1
B
1
B2
B
2
B3
B
3
=
Th
No. of buckets
Network flow
hash
B1
B
1
B2
B
2
B3
B
3
B4
B
4
=
Th
No. of buckets
Network flow
hash
B1
B
2
B2
B
3

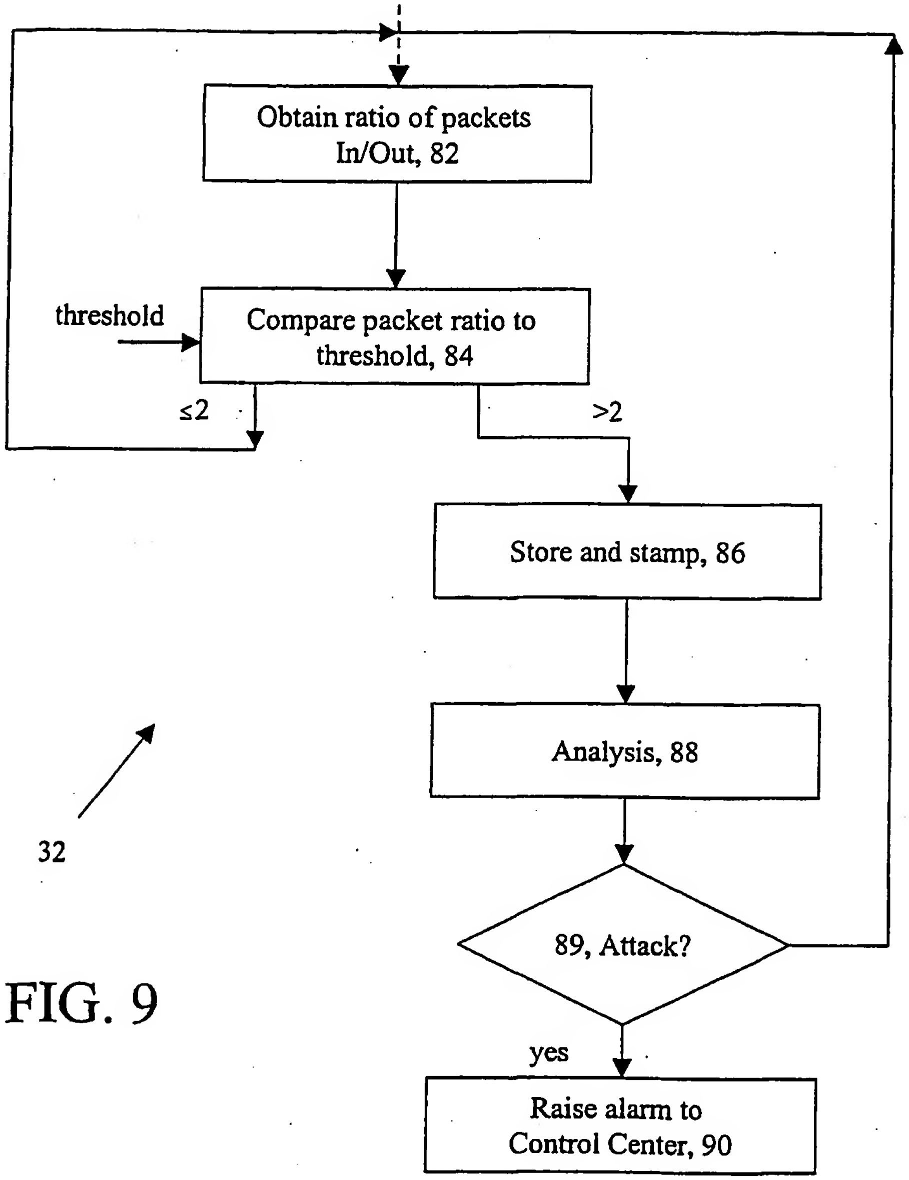

Obtain ratio of packets In/Out, 82
threshold
Compare packet ratio to threshold, 84
≤2
>2
Store and stamp, 86
Analysis, 88
89, Attack?
yes
Raise alarm to Control Center, 90
32

FIG. 9

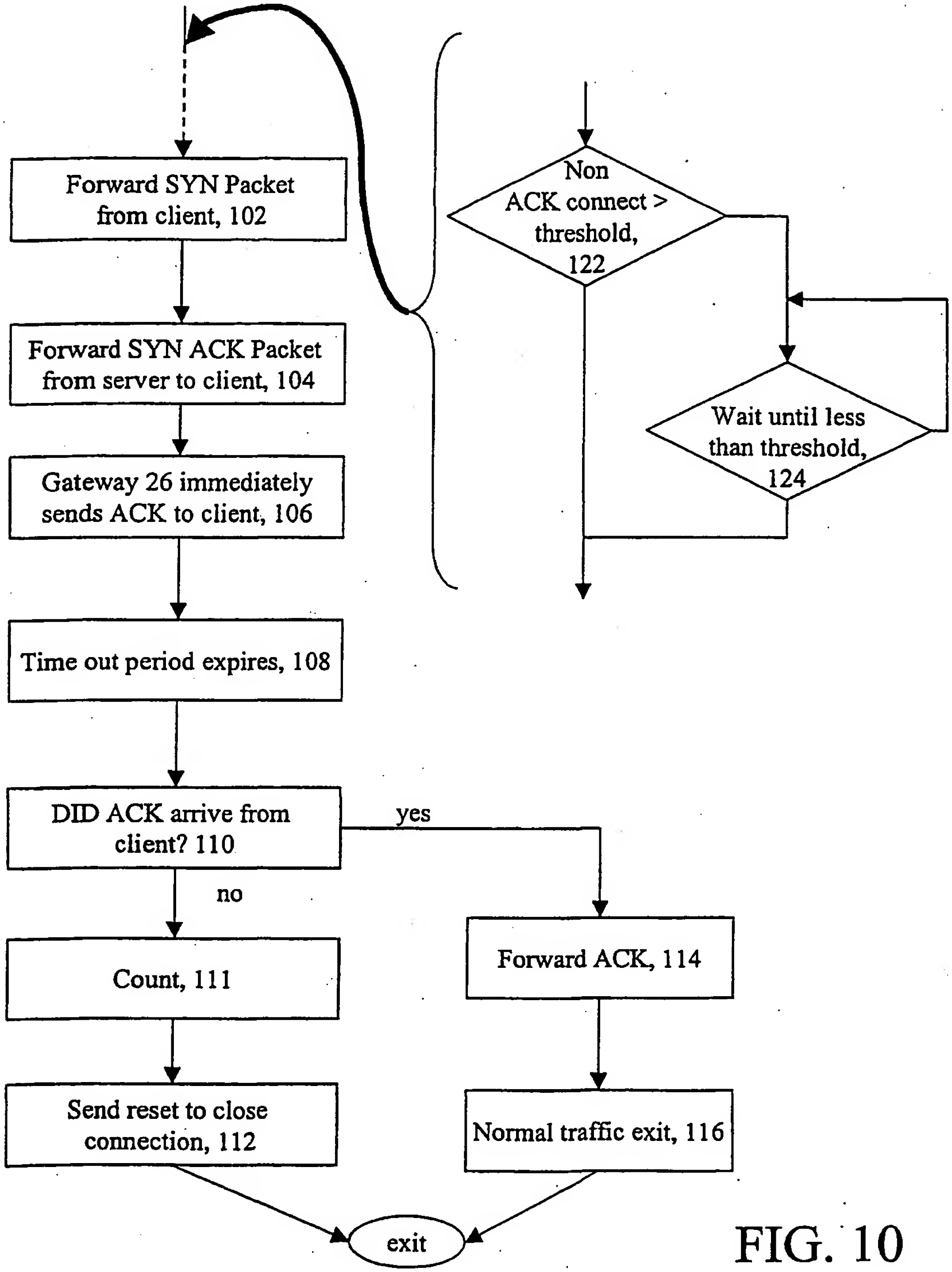

Forward SYN Packet from client, 102
Forward SYN ACK Packet from server to client, 104
Gateway 26 immediately sends ACK to client, 106
Time out period expires, 108
DID ACK arrive from client? 110
yes
no
Count, 111
Send reset to close connection, 112
Forward ACK, 114
Normal traffic exit, 116
exit
Non ACK connect > threshold, 122
Wait until less than threshold, 124

FIG. 10

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US01/27394

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) : G06F 15/16
US CL : 709/235
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 709/235, 232, 233, 234

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

STN
search terms: bucket, monitor, additional, watermark, link

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,854,938 A (OGI) 29 December 1998, col. 8, line 45 - col. 10, line 58.	1-49
Y	US 6,076,112 A (HAUSER et al.) 13 June 2000, see entire document.	1-49
A	US 5,796,956 A (JONES) 18 August 1998, col. 4, line 15 - col. 6, line 54.	1, 14, 21, 22, 32, 35, 45
A	US 6,058,114 A (SETHURAM et al.) 02 May 2000, col. 7, line 26 - col. 9, line 61 and col. 10, line 65 - col. 12, line 14.	1, 14, 21, 22, 32, 35, 45
A	US 6,084,856 A (SIMMONS et al.) 04 July 2000, col. 2, line 25 - col. 3, line 22, col. 4, line 58 - col. 6, line 61 and col. 10, line 65 - col. 12, line 53.	1, 14, 21, 22, 32, 35, 45

□ Further documents are listed in the continuation of Box C. □ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search
29 NOVEMBER 2001

Date of mailing of the international search report
02 JAN 2002

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer Peggy Harrod
JASON D. CARDONE
Telephone No. (703) 305-3800